PHP扩展开发及内核应用

WebrainsBook

Published
with GitBook



目錄

《PHP扩展开发及内核应用》

11年年初因兴趣建立此项目，有幸得到大家关注，万分荣幸。现在把他迁移到github上，以方便更多人参与进来。 项目阅读地址：http://www.walu.cc/phpbook

撰写方法

认领内容

首先, 请在issues中建立一个新的issue, 说明你认领的内容, 和预计完成时间. 然后就可以Fork本项目进行编辑, 等你完成后, 发起一个Pull Request即可.

文件命名

每个章节建立一个md文件，如第11章的第3节，则建立**11.3.md**。

格式规范

请参看已有章节的规范, 要注意的是, 每个章节在底部都需要有一个links节, 包含"目录", "上一节", "下一节"的链接

编译方法

定期同步到phpbook

开始阅读

开始阅读

1 PHP的生命周期

目录

在平常的Web环境中，我们并不需要单独启动PHP，它一般都会作为一个模块自动加载到web-server里面去，如apache加载的php5.so。只要我们启动了web-server，被一起加载的php便会和服务器一起解析被请求的php脚本。

当然，这不是绝对的，当我们以fastcgi模式运行php的时候，往往需要手工通过 命令来启动来启动php后端服务。

links

- 目录
- 下一节: 让我们从SAPI开始

1.1 让我们从SAPI开始

我们平时接触的最多的是web模式下的php，当然你也肯定知道php还有个CLI模式。 其实无论哪种模式，PHP的工作原理都是一样的， 都是作为一种SAPI在运行（Server Application Programming Interface：the API used by PHP to interface with Web Servers）。当我们在终端敲入php这个命令时候，它使用的是"command line sapi"！它就像一个mini的web服务器一样来支持php完成这个请求，请求完成后再重新把控制权交给终端。

简单来说, SAPI就是PHP和外部环境的代理器。它把外部环境抽象后, 为内部的PHP提供一套固定的, 统一的接口, 使得PHP自身实现能够不受错综复杂的外部环境影响，保持一定的独立性

更多内容参看来自Laruence的博客对SAPI的介绍： 深入理解Zend SAPIs

links

- 目录
- 上一节: PHP的生命周期
- 下一节: PHP的启动与终止

1.2 PHP的启动与终止

PHP程序的启动可以看作有两个概念上的启动，终止也有两个概念上的终止。 其中一个是PHP作为Apache(拿它举例，板砖勿扔)的一个模块的启动与终止， 这次启动php会初始化一些必要数据，比如与宿主Apache有关的，并且这些数据是常驻内存的！ 终止与之相对。 还有一个概念上的启动就是当Apache分配一个页面请求过来的时候，PHP会有一次启动与终止，这也是我们最常讨论的一种。

现在我们主要来看一个PHP扩展的生命旅程是怎样走完这四个过程的。

在最初的初始化时候，就是PHP随着Apache的启动而诞生在内存里的时候， 它会把自己所有已加载扩展的MINIT方法(全称Module Initialization，是由每个模块自己定义的函数。)都执行一遍。 在这个时间里，扩展可以定义一些自己的常量、 类、 资源等所有会被用户端的PHP脚本用到的东西。 但你要记住，这里定义的东东都会随着Apache常驻内存，可以被所有请求使用，直到Apache卸载掉PHP模块！

内核中预置了PHP_MINIT_FUNCTION宏函数，来帮助我们实现这个功能：

```
//抛弃作者那个例子，书才看两页整那样的例子太复杂了!
//walu是我扩展的名称
int time_of_minit;//在MINIT()中初始化，在每次页面请求中输出，看看是否变化
PHP_MINIT_FUNCTION(walu)
{
    time_of_minit=time(NULL);//我们在MINIT启动中对他初始化
    return SUCCESS;//返回SUCCESS代表正常，返回FALIURE就不会加载这个扩展了。
}
```

当一个页面请求到来时候，PHP会迅速的开辟一个新的环境，并重新扫描自己的各个扩展， 遍历执行它们各自的RINIT方法(俗称Request Initialization)， 这时候一个扩展可能会初始化在本次请求中会使用到的变量等， 还会初始化等会儿用户端（即PHP脚本）中的变量之类的，内核预置了PHP_RINIT_FUNCTION()这个宏函数来帮我们实现这个功能：

```
int time_of_rinit;//在RINIT里初始化，看看每次页面请求的时候是否变化。
PHP_RINIT_FUNCTION(walu)
{
    time_of_rinit=time(NULL);
    return SUCCESS;
}
```

好了，现在这个页面请求执行的差不多了，可能是顺利的走到了自己文件的最后， 也可能是出师未捷，半道被用户给die或者exit了， 这时候PHP便会启动回收程序，收拾这个请求留下的烂摊子。 它这次会执行所有已加载扩展的RSHUTDOWN（俗称Request Shutdown）方法， 这时候扩展可以抓紧利用内核中的变量表之类的做一些事情， 因为一旦PHP把所有扩展的RSHUTDOWN方法执行完， 便会释放掉这次请求使用过的所有东西， 包括变量表的所有变量、所有在这次请求中申请的内存等等。

内核预置了PHP_RSHUTDOWN_FUNCTION宏函数来帮助我们实现这个功能

```
PHP_RSHUTDOWN_FUNCTION(walu)
{
```

```
    FILE *fp=fopen("time_rshutdown.txt","a+");
    fprintf(fp,"%ld\n",time(NULL));//让我们看看是不是每次请求结束都会在这个文件里追加数据
    fclose(fp);
    return SUCCESS;
}
```

前面该启动的也启动了，该结束的也结束了，现在该Apache老人家歇歇的时候，当Apache通知PHP自己要Stop的时候，PHP便进入MSHUTDOWN（俗称Module Shutdown）阶段。这时候PHP便会给所有扩展下最后通牒，如果哪个扩展还有未了的心愿，就放在自己MSHUTDOWN方法里，这可是最后的机会了，一旦PHP把扩展的MSHUTDOWN执行完，便会进入自毁程序，这里一定要把自己擅自申请的内存给释放掉，否则就杯具了。

内核中预置了PHP_MSHUTDOWN_FUNCTION宏函数来帮助我们实现这个功能：

```
PHP_MSHUTDOWN_FUNCTION(walu)
{
    FILE *fp=fopen("time_mshutdown.txt","a+");
    fprintf(fp,"%ld\n",time(NULL));
    return SUCCESS;
}
```

这四个宏都是在walu.c里完成最终实现的，而他们的则是在/main/php.h里被定义的(其实也是调用的别的宏，本节最后我把这几个宏给展开了，供有需要的人查看)。

好了，现在我们本节内容说完了，下面我们把所有的代码合在一起，并预测一下应该出现的结果：

```
//这些代码都在walu.c里面，不在.h里

int time_of_minit;//在MINIT中初始化，在每次页面请求中输出，看看是否变化
PHP_MINIT_FUNCTION(walu)
{
    time_of_minit=time(NULL);//我们在MINIT启动中对他初始化
    return SUCCESS;
}

int time_of_rinit;//在RINIT里初始化，看看每次页面请求的时候是否变化。
PHP_RINIT_FUNCTION(walu)
{
    time_of_rinit=time(NULL);
    return SUCCESS;
}

PHP_RSHUTDOWN_FUNCTION(walu)
{
    FILE *fp=fopen("/cnan/www/erzha/time_rshutdown.txt","a+");//请确保文件可写，否则apache会
    fprintf(fp,"%d\n",time(NULL));//让我们看看是不是每次请求结束都会在这个文件里追加数据
    fclose(fp);
    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(walu)
{
    FILE *fp=fopen("/cnan/www/erzha/time_mshutdown.txt","a+");//请确保文件可写，否则apache会
```

```
    fprintf(fp,"%d\n",time(NULL));
    return SUCCESS;
}

//我们在页面里输出time_of_minit和time_of_rinit的值
PHP_FUNCTION(walu_test)
{
    php_printf("%d&lt;br /&gt;",time_of_minit);
    php_printf("%d&lt;br /&gt;",time_of_rinit);
    return;
}
```

- time_of_minit的值每次请求都不变。
- time_of_rinit的值每次请求都改变。
- 每次页面请求结束都会往time_rshutdown.txt中写入数据。
- 只有在apache结束后time_mshutdown.txt才写入有数据。

> 多谢 闸北陆小洪 指出的有关time_of_rinit的笔误。

上面便是PHP中典型的启动-终止模型，实际情况可能因为模式不同而有所变化， 到底PHP的启动-终止会有多少种不同变化方式，请看下一节。

links

- 目录
- 上一节: 让我们从SAPI开始
- 下一节: PHP的生命周期

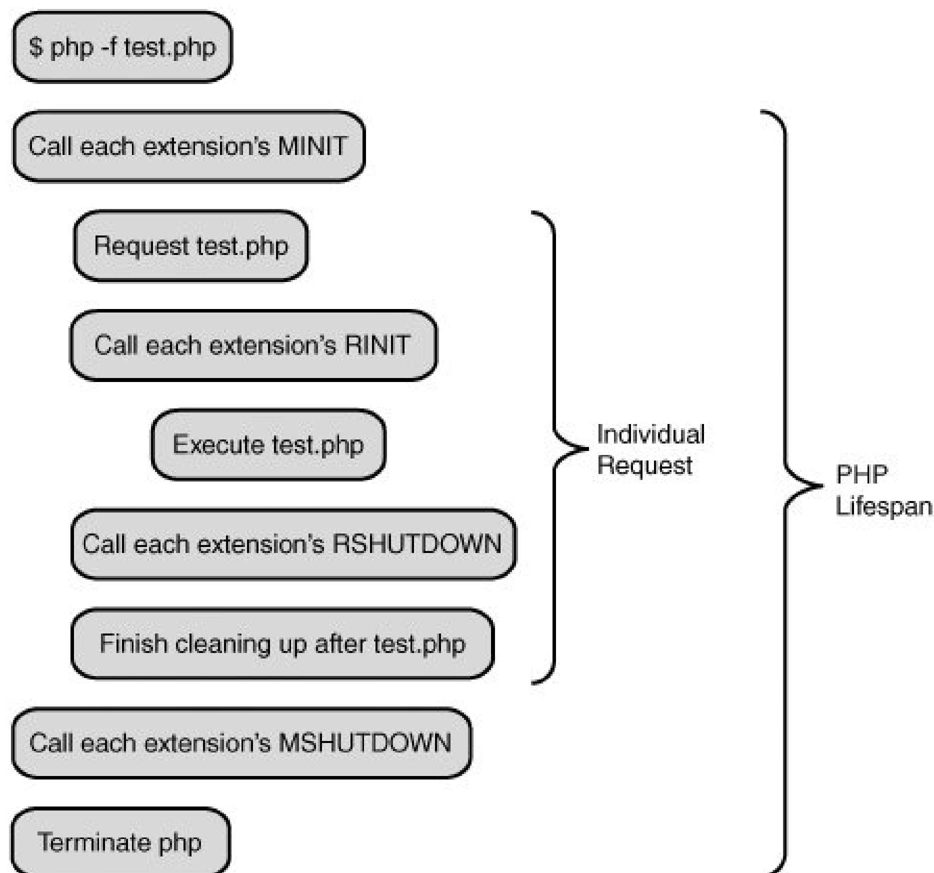
1.3 PHP的生命周期

一个PHP实例，无论通过http请求调用的，还是从命令行启动的，都会向我们上一节说的那样， 依次进行Module init、Request init、Request Shutdown、Module shutdown四个过程， 当然之间还会执行脚本自己的逻辑。 那么两种init和两种shutdown各会执行多少次、各自的执行频率有多少呢？ 这取决与PHP是用什么sapi与宿主通信的。最常见的四种方式如下所列：

- 直接以CLI/CGI模式调用
- 多进程模式
- 多线程模式
- Embedded(嵌入式，在自己的C程序中调用Zend Engine)

1、 CLI/CGI

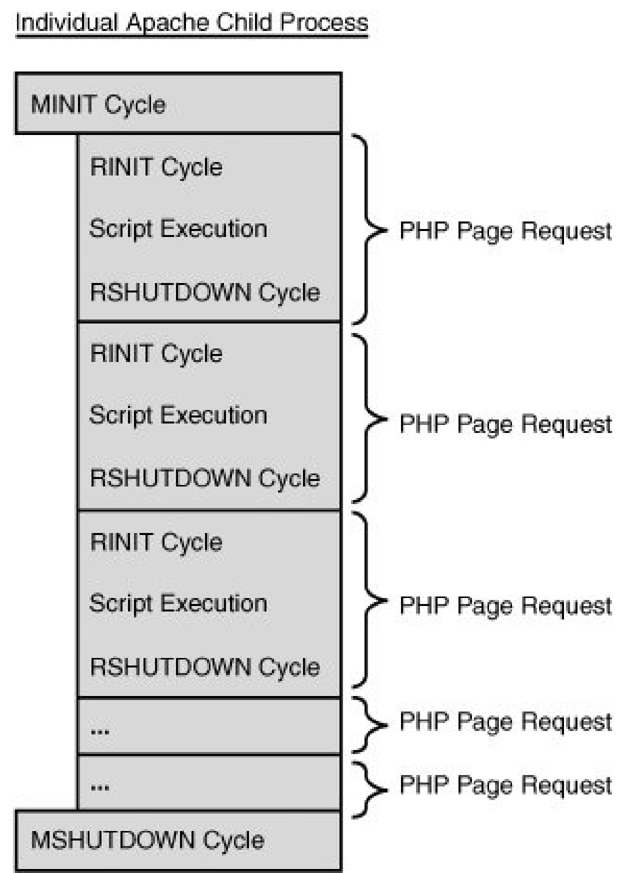
CLI和CGI的SAPI是相当特殊的，因为这时PHP的生命周期完全在一个单独的请求中完成。虽然简单，不过我们以前提过的两种init和两种shutdown仍然都会被执行。图1.1展示了PHP在这种模式下是怎么工作的。



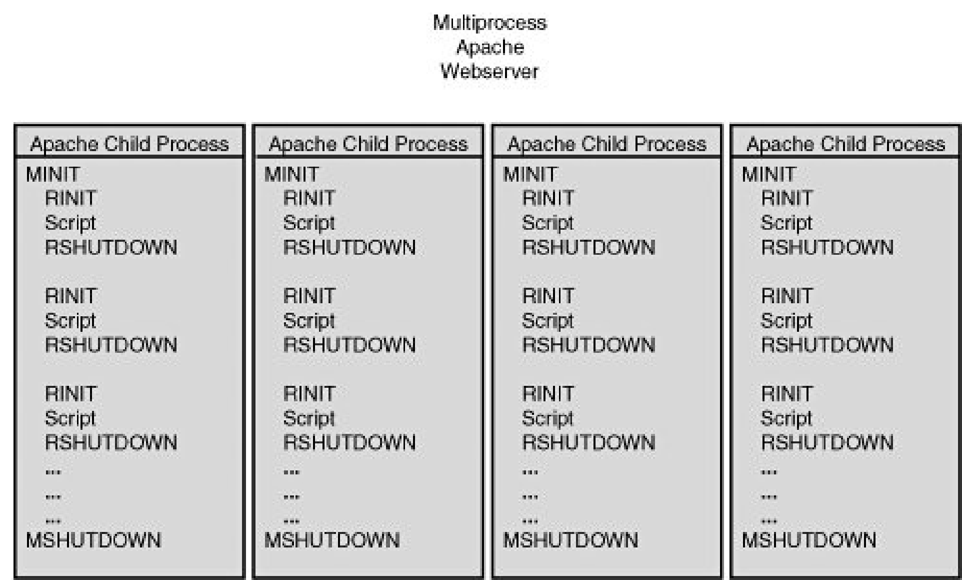
2、 多进程模式

[ps:书是**2006**年出版的，所以你应该理解作者说多进程是主流**]** PHP最常见的工作方式便是编译成为Apache2 的Pre-fork MPM或者Apache1 的APXS 模式，其它web服务器也大多用相同的方式工作，在本书后面，把这种方式统一叫做多进程方式。 给它起这个名字是有原因的，不是随便拍拍屁股拍拍脑袋定下来的。 当Apache启动的时候，会立即把自己fork出好几个子进程，每一个进程都有自己独立的内存空间， 也

就代表了有自己独立的变量、函数等。在每个进程里的PHP的工作方式如下图所示：



因为是fork出来的，所以各个进程间的数据是彼此独立，不会受到外界的干扰(ps：fork后可以用管道等方式实现进程间通信)。 这是一片独立天地，它允许每个子进程做任何事情，玩七十码、躲猫猫都没人管，办公室拿砍刀玩自杀也没事， 下图展示了从apache的视角来看多进程工作模式下的PHP：



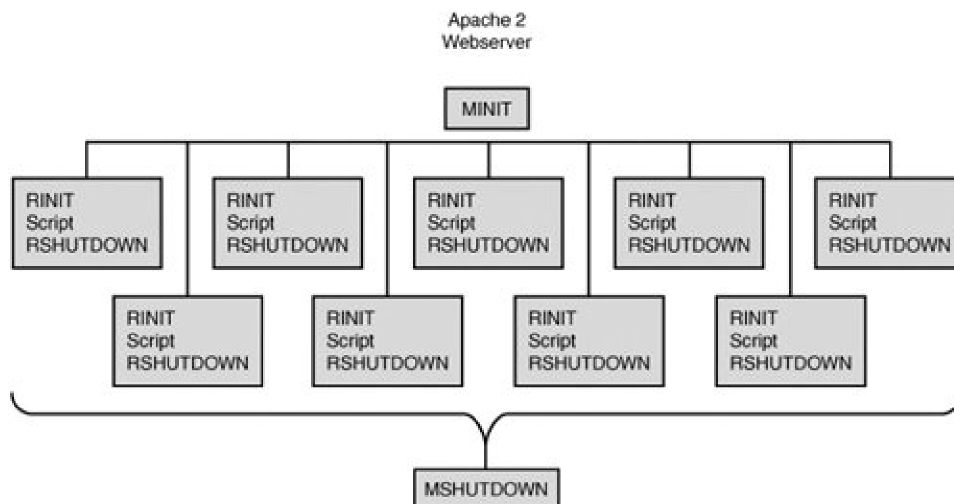
3、多线程模式

随着时代的进步，PHP越来越多的在多线程模式下工作，就像IIS的isapi和Apache MPM worker(支持混合的多线程多进程的多路处理模块)。 在这种模式下，只有一个服务器进程在运行着，但会同时运行很多线

程，这样可以减少一些资源开销，像Module init和Module shutdown就只需要运行一次就行了，一些全局变量也只需要初始化一次，因为线程独具的特质，使得各个请求之间方便的共享一些数据成为可能。

> 其实多线程与MINIT、MSHUTDOWN只执行一次并没有什么联系，多进程模式下一样可以实现。

下图展示了在这种模式下PHP的工作流程：



4、Embed

Embed SAPI是一种比较特殊的sapi，容许你在C/C++语言中调用PHP/ZE提供的函数。并且这种sapi和上面的三种一样，按Module Init、Request Init、Rshutdown、mshutdown的流程执行着。当然，这只是其中一种情况。因为特定的应用有自己特殊的需求，只是在处理PHP脚本这个环节基本一致。

真正令emebed模式独特的是因为它可能随时嵌入到某个程序里面去(比如你的**test.exe**里)，然后被当作脚本的一部分在一个请求的时候执行。控制权在PHP和原程序间来回传递。关于嵌入式的PHP在第20章会有应用，到时我们再用实例介绍这个不经常使用的sapi。

关于Embed SAPI应用的文章

- Laruence大哥的使用PHP Embed SAPI实现Opcodes查看器

links

- 目录
- 上一节 PHP的启动与终止
- 下一节 线程安全

1.4 线程安全

在PHP初期，是作为单进程的CGI来运行的，所以并没有考虑线程安全问题。 我们可以随意的在全局作用域中设置变量并在程序中对他进行修改、访问，内核申请的资源如果没有正确的释放， 也会在CGI进程结束后自动地被清理干净。

后来，php被作为apache多进程模式下的一个模块运行，但是这仍然把php局限在一个进程里， 我们设置的全局变量，只要在每个请求之前将其正确的初始化，并在每个请求之后正确的清理干净， 便不会带来什么麻烦。由于对于一个进程来说，同一个时间只能处理一个请求， 所以这是内核中加入了针对每个请求的内存管理功能，来防止服务器资源利用出现错误。

随着使用在多线程模式的软件系统越来越多，php内核中亟需一种新的资源管理方式， 并最终在php内核中形成了一个新的抽象层：TSRM(Thread Safe Resource Management)。

线程安全与非线程安全

在一个没有线程的程序中，我们往往倾向于把全局变量声明在源文件的顶部， 编辑器会自动的为它分配资源供我们在声明语句之下的程序逻辑中使用。

（即使通过**fork()**出一个子进程，它也会重新申请一段内存，父子进程中的变量从此没有了任何联系）

但是在一个多线程的程序中，如果我们需要每个线程都拥有自己独立的资源的话， 便需要为每个线程独立开辟出一个区域来存放它们各自的资源， 在使用资源的时候，每个线程便会只在自己的那一亩三分地里找，而不会拔了别人的庄稼。

Thread-Safe Data Pools(线程安全的资源池？)

在扩展的Module Init里，扩展可以调用ts_allocate_id()来告诉TRSM自己需要多少资源。 TRSM接收后更新系统使用的资源，并得到一个指向刚分配的那份资源的id。

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
int sample_globals_id;

PHP_MINIT_FUNCTION(sample)
{
    ts_allocate_id(&sample_globals_id,
        sizeof(php_sample_globals),
        (ts_allocate_ctor) php_sample_globals_ctor,
        (ts_allocate_dtor) php_sample_globals_dtor);
    return SUCCESS;
}
```

当一个请求需要访问数据段的时候，扩展从TSRM层请求当前线程的资源池， 以ts_allocate_id()返回的资源ID来获取偏移量。 换句话说，在代码流中，你可能会在前面所说的MINIT语句中碰到

SAMPLE_G(sampleint) = 5; 这样的语句。在线程安全的构建下，这个语句通过一些宏扩展如下：

```
(((php_sample_globals*)(*((void ***)tsrm_ls))[sample_globals_id-1])->sampleint = 5;
```

如果你看不懂上面的转换也不用沮丧，它已经很好的封装在PHPAPI中了，以至于许多开发者都不需要知道它怎样工作的。

当不在线程环境时

因为在PHP的线程安全构建中访问全局资源涉及到在线程数据池查找对应的偏移量，这是一些额外的负载，结果就是它比对应的非线程方式（直接从编译期已经计算好的真实的全局变量地址中取出数据）慢一些。 考虑上面的例子，这一次在非线程构建下：

```
typedef struct {
    int sampleint;
    char *samplestring;
} php_sample_globals;
php_sample_globals sample_globals;

PHP_MINIT_FUNCTION(sample)
{
    php_sample_globals_ctor(&sample_globals TSRMLS_CC);
    return SUCCESS;
}
```

首先注意到的是这里并没有定义一个int型的标识去引用全局的结构定义， 只是简单的在进程的全局空间定义了一个结构体。 也就是说SAMPLE_G(sampleint) = 5;展开后就是sample_globals.sampleint = 5; 简单，快速，高效。 非线程构建还有进程隔离的优势，这样给定的请求碰到完全出乎意料的情况时，它也不会影响其他进程， 即便是产生段错误也不会导致整个webserver瘫痪。 实际上，Apache的MaxRequestsPerChild指令就是设计用来提升这个特性的， 它经常性的有目的性的kill掉子进程并产生新的子进程，来避免某些可能由于进程长时间运行“累积”而来的问题（比如内存泄露）。

访问全局变量

在创建一个扩展时，你并不知道它最终的运行环境是否是线程安全的。幸运的是，你要使用的标准包含文件集合中已经包含了条件定义的ZTS预处理标记。当PHP因为SAPI需要或通过enable-maintainer-zts选项安装等原因以线程安全方式构建时，这个值会被自动的定义，并可以用一组#ifdef ZTS这样的指令集去测试它的值。 就像你前面看到的，只有在PHP以线程安全方式编译时，才会存在线程安全池，只有线程安全池存在时，才会真的在线程安全池中分配空间。这就是为什么前面的例子包裹在ZTS检查中的原因，非线程方式供非线程构建使用。 在本章前面PHP_MINIT_FUNCTION(myextension)的例子中，你可以看到#ifdef ZTS被用作条件调用正确的全局初始代码。对于ZTS模式它使用ts_allocate_id()弹出myextension_globals_id变量，而非ZTS模式只是直接调用myextension_globals的初始化方法。这两个变量已经在你的扩展源文件中使用Zend宏：DECLARE_MODULE_GLOBALS(myextension)声明，它将自动的处理对ZTS的测试并依赖构建的ZTS模式选择正确的方式声明。 在访问这些全局变量的时候，你需要使用前面给出的自定义宏SAMPLE_G()。在第12章，你将学习到怎样设计这个宏以使它可以依赖ZTS模式自动展开。

即便你不需要线程也要考虑线程

正常的PHP构建默认是关闭线程安全的，只有在被构建的sapi明确需要线程安全或线程安全在./configure阶段显式的打开时，才会以线程安全方式构建。 给出了全局查找的速度问题和进程隔离的缺点后，你可能会疑惑为什么明明不需要还有人故意打开它呢？这是因为，多数情况下，扩展和SAPI的开发者认为你是线程安全开关的操作者，这样做可以很大程度上确保新代码可以在所有环境中正常运行。 当线程安全启用时，一个名为tsrm_ls的特殊指针被增加到了很多的内部函数原型中。这个指针允许PHP区分不同线程的数据。回想一下本章前面ZTS模式下的SAMPLE_G()宏函数中就使用了它。没有它，正在执行的函数就不知道查找和设置哪个线程的符号表；不知道应该执行哪个脚本，引擎也完全无法跟踪它的内部寄存器。这个指针保留了线程处理的所有页面请求。 这个可选的指针参数通过下面一组定义包含到原型中。当ZTS禁用时，这些定义都被展开为空；当ZTS开启时，它们展开如下：

```
#define TSRMLS_D     void ***tsrm_ls
#define TSRMLS_DC     , void ***tsrm_ls
#define TSRMLS_C     tsrm_ls
#define TSRMLS_CC     , tsrm_ls
```

非ZTS构建对下面的代码看到的是两个参数：int, char 。在*ZTS*构建下，原型则包含三个参数：*int, char* , void *。当你的程序调用这个函数时，只有在ZTS启用时才需要传递第三个参数。下面代码的第二行展示了宏的展开：

```
int php_myext_action(int action_id, char *message TSRMLS_DC);
php_myext_action(42, "The meaning of life" TSRMLS_CC);
```

通过在函数调用中包含这个特殊的变量，php_myext_action就可以使用tsrm_ls的值和MYEXT_G()宏函数一起访问它的线程特有全局数据。在非ZTS构建上，tsrm_ls将不可用，但是这是ok的，因为此时MYEXT_G()宏函数以及其他类似的宏都不会使用它。 现在考虑，你在一个新的扩展上工作，并且有下面的函数，它可以在你本地使用CLI SAPI的构建上正常工作，并且即便使用apache 1的apxs SAPI编译也可以正常工作：

```
static int php_myext_isset(char *varname, int varname_len)
{
    zval **dummy;


    if (zend_hash_find(EG(active_symbol_table),
        varname, varname_len + 1,
        (void**)&dummy) == SUCCESS) {
        /* Variable exists */
        return 1;
    } else {
        /* Undefined variable */
        return 0;
    }
}
```

所有的一切看起来都工作正常，你打包这个扩展发送给他人构建并运行在生产服务器上。让你气馁的是，对方报告扩展编译失败。 事实上它们使用了Apache 2.0的线程模式，因此它们的php构建启用了ZTS。当

编译期碰到你使用的EG()宏函数时，它尝试在本地空间查找tsrm_ls没有找到，因为你并没有定义它并且没有在你的函数中传递。 修复这个问题非常简单；只需要在php_myext_isset()的定义上增加TSRMLS_DC，并在每行调用它的地方增加TSRMLS_CC。不幸的是，现在对方已经有点不信任你的扩展质量了，这样就会推迟你的演示周期。这种问题越早解决越好。 现在有了enable-maintainer-zts指令。通过在./configure时增加该指令来构建php，你的构建将自动的包含ZTS，哪怕你当前的SAPI（比如CLI）不需要它。打开这个开关，你可以避免这些常见的不应该出现的错误。 注意：在PHP4中，enable-maintainer-zts标记等价的名字是enable-experimental-zts；请确认使用你的php版本对应的正确标记。

寻回丢失的tsrm_ls

有时，我们需要在一个函数中使用tsrm_ls指针，但却不能传递它。通常这是因为你的扩展作为某个使用回调的库的接口，它并没有提供返回抽象指针的地方。考虑下面的代码片段：

```
void php_myext_event_callback(int eventtype, char *message)
{
    zval *event;


    /* $event = array('event'=>$eventtype,
                    'message'=>$message) */
    MAKE_STD_ZVAL(event);
    array_init(event);
    add_assoc_long(event, "type", eventtype);
    add_assoc_string(event, "message", message, 1);


    /* $eventlog[] = $event; */
    add_next_index_zval(EXT_G(eventlog), event);
}
PHP_FUNCTION(myext_startloop)
{
    /* The eventlib_loopme() function,
     * exported by an external library,
     * waits for an event to happen,
     * then dispatches it to the
     * callback handler specified.
     */
    eventlib_loopme(php_myext_event_callback);
}
```

虽然你可能不完全理解这段代码，但你应该注意到了回调函数中使用了EXT_G()宏函数，我们知道在线程安全构建下它需要tsrm_ls指针。修改函数原型并不好也不应该这样做，因为外部的库并不知道php的线程安全模型。那这种情况下怎样让tsrm_ls可用呢？ 解决方案是前面提到的名为TSRMLS_FETCH()的Zend宏函数。将它放到代码片段的顶部，这个宏将执行给予当前线程上下文的查找，并定义本地的tsrm_ls指针拷贝。 这个宏可以在任何地方使用并且不用通过函数调用传递tsrm_ls，尽管这看起来很诱人，但是，要注意到这一点：TSRMLS_FETCH调用需要一定的处理时间。这在单次迭代中并不明显，但是随着你的线程数增多，随着你调用TSRMLS_FETCH()的点的增多，你的扩展就会显现出这个瓶颈。因此，请谨慎的使用它。 注意：为了和c++编译器兼容，请确保将TSRMLS_FETCH()和所有变量定义放在给定块作用域的顶部（任何其他语句之前）。因为TSRMLS_FETCH()宏自身有多种不同的解析方式，因此最好将它作为变量定义的最后一行。

links

- 目录
- 上一节: PHP的生命周期
- 下一节: 小结

1.5 PHP的生命周期

这一章讲述了一些后续章节需要的基础概念，是你编写优质的PHP扩展的基础。

links

- 目录
- 上一节: 线程安全
- 下一节: PHP变量在内核中的实现

2 PHP变量在内核中的实现

所有的编程语言都要提供一种数据的存储与检索机制，PHP也不例外。其它语言大都需要在使用变量之前先定义，并且它的类型也是无法再次改变的，而PHP却允许程序猿自由的使用变量而无须提前定义，甚至可以随时随意的对已存在的变量转换成其它任何PHP支持的数据类型。在程序在运行的时候，PHP还会自动的根据需求转换变量的类型。

我认为阅读本书的人都已经是标准的PHP程序猿了，所以你们也肯定体验过PHP的弱类型的变量体系。众所周知，PHP引擎是用C写的，而C确实一种强类型的编程语言，PHP内核中是如何用C来实现自己的这种弱类型特性的，你将在本章中找到答案！

links

- 目录
- 1.5 小结
- 2.1 变量的类型

2.1 变量的类型

PHP在内核中是通过zval这个结构体来存储变量的，它的定义在Zend/zend.h文件里，简短精炼，只有四个成员组成：

```
struct _zval_struct {
    zvalue_value value;    /* 变量的值 */
    zend_uint refcount__gc;
    zend_uchar type;    /* 变量当前的数据类型 */
    zend_uchar is_ref__gc;
};
typedef struct _zval_struct zval;

//在Zend/zend_types.h里定义的：
typedef unsigned int zend_uint;
typedef unsigned char zend_uchar;
```

zval里的refcount**gc是zend_uint**类型，也就是**unsigned int**型，**is_ref**gc和type则是unsigned char型的。

保存变量值的value则是zvalue_value类型(PHP5)，它是一个union，同样定义在了Zend/zend.h文件里：

```
typedef union _zvalue_value {
    long lval;    /* long value */
    double dval;    /* double value */
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;    /* hash table value */
    zend_object_value obj;
} zvalue_value;
```

在以上实现的基础上，PHP语言得以实现了8种数据类型，这些数据类型在内核中的分别对应于特定的常量，它们分别是：

常量名称：	
IS_NULL	第一次使用的变量如果没有初始化过，则会自动的被赋予这个常量，当然我们也可以在PHP语言中通过null这个常量来给予变量null类型的值。 这个类型的值只有一个 ，就是NULL，它和0与false是不同的。
IS_BOOL	布尔类型的变量有两个值，true或者false。在PHP语言中，while、if等语句会自动的把表达式的值转成这个类型的。
IS_LONG	PHP语言中的整型，在内核中是通过所在操作系统的signed long数据类型来表示的。 在最常见的32位操作系统中，它可以存储从-2147483648 到 +2147483647范围内的任一整数。 有一点需要注意的是，如果PHP语言中的整型变量超出最大值或者最小值，它并不会直接溢出， 而是会被内核转换成IS_DOUBLE类型的值然后再参与计算。 再者，因为使用了signed long来作为载体，所以这也就解释了为什么PHP语言中的整型数据都是带符号的了。 ```c $a=2147483647; $a++; echo $a;//会正确的输出 2147483648； ````
	PHP中的浮点数据是通过C语言中的signed double型变量来存储的， 这最终取决与所在操作系统的浮点型实现。 我们做为程序猿，应该知道计算机是无法精准的表示浮点数的， 而是采用了科学计数法来保存某个精度的浮点数。 用科学计数法，计算机只用8位便可以保存2.225x10^(-308)~~1.798x10^308之间的浮点数。 用计算机来处理浮点数简直就是一场噩梦，十

IS_DOUBLE	进制的0.5转成二进制是0.1， 0.8转换后是0.1100110011....。 但是当我们从二进制转换回来的时候，往往会发现并不能得到0.8。 我们用1除以3这个例子来解释这个现象：1/3=0.3333333333.....，它是一个无限循环小数， 但是计算机可能只能精确存储到0.333333，当我们再乘以三时， 其实计算机计算的数是0.333333*3=0.999999，而不是我们平时数学中所期盼的1.0.
IS_STRING	PHP中最常用的数据类型——字符串，在内存中的存储和C差不多， 就是一块能够放下这个变量所有字符的内存，并且在这个变量的zval实现里会保存着指向这块内存的指针。 与C不同的是，PHP内核还同时在zval结构里保存着这个字符串的实际长度， 这个设计使PHP可以在字符串中嵌入'\0'字符，也使PHP的字符串是二进制安全的， 可以安全的存储二进制数据！本着艰苦朴素的作风，内核只会为字符串申请它长度+1的内存， 最后一个字节存储的是'\0'字符，所以在不需要二进制安全操作的时候， 我们可以像通常C语言的方式那样来使用它。
IS_ARRAY	数组是一个非常特殊的数据类型，它唯一的功能就是聚集别的变量。 在C语言中，一个数组只能承载一种类型的数据，而PHP语言中的数组则灵活的多， 它可以承载任意类型的数据，这一切都是HashTable的功劳， 每个HashTable中的元素都有两部分组成：索引与值， 每个元素的值都是一个独立的zval（确切的说应该是指向某个zval的指针）。
IS_OBJECT	和数组一样，对象也是用来存储复合数据的，但是与数组不同的是， 对象还需要保存以下信息：方法、访问权限、类常量以及其它的处理逻辑。 相对与zend engine V1，V2中的对象实现已经被彻底修改， 所以我们PHP扩展开发者如果需要自己的扩展支持面向对象的工作方式， 则应该对PHP5和PHP4分别对待！
IS_RESOURCE	有一些数据的内容可能无法直接呈现给PHP用户的， 比如与某台mysql服务器的链接，或者直接呈现出来也没有什么意义。 但用户还需要这类数据，因此PHP中提供了一种名为Resource(资源)的数据类型。 有关这个数据类型的事宜将在第九章中介绍，现在我们只要知道有这么一种数据类型就行了。

zval结构体里的type成员的值便是以上某个IS_*常量之一。 内核通过检测变量的这个成员值来知道他是什么类型的数据并做相应的后续处理。

如果要我们检测一个变量的类型，最直接的办法便是去读取它的type成员的值：

```
void describe_zval(zval *foo)
{
    if (foo->type == IS_NULL)
    {
        php_printf("这个变量的数据类型是： NULL");
    }
    else
    {
        php_printf("这个变量的数据类型不是NULL，这种数据类型对应的数字是： %d", foo->type);
    }
}
```

虽然上述实现是正确的，但我们强烈建议你不要这样做。

PHP内核以后可能会修改变量的实现方式，所以检测type的方法可能在以后就不能用了。 为了解决这个兼容问题，zend头文件中定义了大量的宏，供我们检测、操作变量使用， 使用这些宏不但让我们的程序更易读，还具有更好的兼容性。 这里我们用Z_TYPE_P()宏来改写上面那个程序。

```
void describe_zval(zval *foo)
{
    if ( Z_TYPE_P(foo) == IS_NULL )
    {
        php_printf("这个变量的数据类型是： NULL");
    }
    else
    {
```

```
        php_printf("这个变量的数据类型不是NULL，这种数据类型对应的数字是： %d", Z_TYPE_P(foo));
    }
}
```

php_printf()函数是内核对**printf()**函数的一层封装，我们可以像使用**printf()**函数那样使用它。

以一个P结尾的宏的参数大多是*zval*型变量。 此外获取变量类型的宏还有两个，分别是*Z_TYPE*和*Z_TYPE_PP*，前者的参数是*zval*型，而后者的参数则是*zval。

这样我们便可以猜测一下php内核是如何实现gettype这个函数了，代码如下：

```
//开始定义php语言中的函数gettype
PHP_FUNCTION(gettype)
{
    //arg间接指向调用gettype函数时所传递的参数。是一个zval**结构
    //所以我们要对他使用__PP后缀的宏。
    zval **arg;

    //这个if的操作主要是让arg指向参数～
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "Z", &arg) == FAILURE) {
        return;
    }

    //调用Z_TYPE_PP宏来获取arg指向zval的类型。
    //然后是一个switch结构，RETVAL_STRING宏代表这gettype函数返回的字符串类型的值
    switch (Z_TYPE_PP(arg)) {
        case IS_NULL:
            RETVAL_STRING("NULL", 1);
            break;

        case IS_BOOL:
            RETVAL_STRING("boolean", 1);
            break;

        case IS_LONG:
            RETVAL_STRING("integer", 1);
            break;

        case IS_DOUBLE:
            RETVAL_STRING("double", 1);
            break;

        case IS_STRING:
            RETVAL_STRING("string", 1);
            break;

        case IS_ARRAY:
            RETVAL_STRING("array", 1);
            break;

        case IS_OBJECT:
            RETVAL_STRING("object", 1);
            break;

        case IS_RESOURCE:
```

```
                {
                    char *type_name;
                    type_name = zend_rsrc_list_get_rsrc_type(Z_LVAL_PP(arg) TSRMLS_CC);
                    if (type_name) {
                        RETVAL_STRING("resource", 1);
                        break;
                    }
                }

            default:
                RETVAL_STRING("unknown type", 1);
        }
    }
```

以上三个宏的定义在Zend/zend_operators.h里，定义分别是：

```
#define Z_TYPE(zval)        (zval).type
#define Z_TYPE_P(zval_p)    Z_TYPE(*zval_p)
#define Z_TYPE_PP(zval_pp)    Z_TYPE(**zval_pp)
```

links

- 目录
- 2 第二章目录
- 2.2 变量的值

2.2 变量的值

PHP内核提供了三个基础宏来方便我们对变量的值进行操作，这几个宏同样以Z_开头，并且P结尾和PP结尾的同上一节中的宏一样，分别代表这参数是指针还是指针的指针。

此外，为了进一步方便我们的工作，内核中针对具体的数据类型分别定义了相应的宏。如针对IS_BOOL型的BVAL组合(Z_BVAL、Z_BVAL_P、Z_BVAL_PP)和针对IS_DOUBLE的DVAL组合(Z_DVAL、ZDVAL_P、ZDVAL_PP)等等。我们通过下面这个例子来应用一下这几个宏:

```
void display_value(zval zv,zval *zv_p,zval **zv_pp)
{
    if( Z_TYPE(zv) == IS_NULL )
    {
        php_printf("类型是 IS_NULL!\n");
    }

    if( Z_TYPE_P(zv_p) == IS_LONG )
    {
        php_printf("类型是 IS_LONG，值是：%ld" , Z_LVAL_P(zv_p));
    }

    if(Z_TYPE_PP(zv_pp) == IS_DOUBLE )
    {
        php_printf("类型是 IS_DOUBLE,值是：%f" , Z_DVAL_PP(zv_pp) );
    }
}
```

string型变量比较特殊，因为内核在保存String型变量时，不仅保存了字符串的值，还保存了它的长度，所以它有对应的两种宏组合STRVAL和STRLEN，即：Z_STRVAL、Z_STRVAL_P、Z_STRVAL_PP与Z_STRLEN、Z_STRLEN_P、Z_STRLEN_PP。前一种宏返回的是char *型，即字符串的地址；后一种返回的是int型，即字符串的长度。

```
void display_string(zval *zstr)
{
    if (Z_TYPE_P(zstr) != IS_STRING) {
        php_printf("这个变量不是字符串!\n");
        return;
    }
    PHPWRITE(Z_STRVAL_P(zstr), Z_STRLEN_P(zstr));
    //这里用了PHPWRITE宏，只要知道它是从Z_STRVAL_P(zstr)地址开始，输出Z_STRLEN_P(zstr)长度的字符就
}
```

Array型变量的值其实是存储在C语言实现的HashTable中的，我们可以用ARRVAL组合宏（Z_ARRVAL, Z_ARRVAL_P, Z_ARRVAL_PP）这三个宏来访问数组的值。如果你看旧版本php的源码或者部分pecl扩展的源码，可能会发现一个HASH_OF()宏，这个宏等价于Z_ARRVAL_P()。但不推荐在新代码中再使用了。

对象是一个复杂的结构体（zend_object_value结构体），不仅存储属性的定义、属性的值，还存储着访问权限、方法等信息。内核中定义了以下组合宏让我们方便的操作对象：OBJ_HANDLE：返回handle标识

符， OBJ_HT：handle表， OBJCE：类定义， OBJPROP：HashTable的属性， OBJ_HANDLER：在OBJ_HT中操作一个特殊的handler方法。 现在不用担心这些宏对象的意思，后续有专门的章节介绍object。

资源型变量的值其实就是一个整数，可以用RESVAL组合宏来访问它，我们把它的值传给zend_fetch_resource函数，便可以得到这个资源的操作句柄，如mysql的链接句柄等。有关资源的内容我们将在第9章展开叙述。

有关值操作的宏都定义在./Zend/zend_operators.h文件里：

```
//操作整数的
#define Z_LVAL(zval)            (zval).value.lval
#define Z_LVAL_P(zval_p)        Z_LVAL(*zval_p)
#define Z_LVAL_PP(zval_pp)        Z_LVAL(**zval_pp)

//操作IS_BOOL布尔型的
#define Z_BVAL(zval)            ((zend_bool)(zval).value.lval)
#define Z_BVAL_P(zval_p)        Z_BVAL(*zval_p)
#define Z_BVAL_PP(zval_pp)        Z_BVAL(**zval_pp)

//操作浮点数的
#define Z_DVAL(zval)            (zval).value.dval
#define Z_DVAL_P(zval_p)        Z_DVAL(*zval_p)
#define Z_DVAL_PP(zval_pp)        Z_DVAL(**zval_pp)

//操作字符串的值和长度的
#define Z_STRVAL(zval)            (zval).value.str.val
#define Z_STRVAL_P(zval_p)        Z_STRVAL(*zval_p)
#define Z_STRVAL_PP(zval_pp)        Z_STRVAL(**zval_pp)

#define Z_STRLEN(zval)            (zval).value.str.len
#define Z_STRLEN_P(zval_p)        Z_STRLEN(*zval_p)
#define Z_STRLEN_PP(zval_pp)        Z_STRLEN(**zval_pp)

#define Z_ARRVAL(zval)            (zval).value.ht
#define Z_ARRVAL_P(zval_p)        Z_ARRVAL(*zval_p)
#define Z_ARRVAL_PP(zval_pp)        Z_ARRVAL(**zval_pp)

//操作对象的
#define Z_OBJVAL(zval)            (zval).value.obj
#define Z_OBJVAL_P(zval_p)        Z_OBJVAL(*zval_p)
#define Z_OBJVAL_PP(zval_pp)        Z_OBJVAL(**zval_pp)

#define Z_OBJ_HANDLE(zval)        Z_OBJVAL(zval).handle
#define Z_OBJ_HANDLE_P(zval_p)        Z_OBJ_HANDLE(*zval_p)
#define Z_OBJ_HANDLE_PP(zval_p)        Z_OBJ_HANDLE(**zval_p)

#define Z_OBJ_HT(zval)            Z_OBJVAL(zval).handlers
#define Z_OBJ_HT_P(zval_p)        Z_OBJ_HT(*zval_p)
#define Z_OBJ_HT_PP(zval_p)        Z_OBJ_HT(**zval_p)

#define Z_OBJCE(zval)            zend_get_class_entry(&(zval) TSRMLS_CC)
#define Z_OBJCE_P(zval_p)        Z_OBJCE(*zval_p)
#define Z_OBJCE_PP(zval_pp)        Z_OBJCE(**zval_pp)

#define Z_OBJPROP(zval)            Z_OBJ_HT((zval))->get_properties(&(zval) TSRMLS_CC)
```

```
#define Z_OBJPROP_P(zval_p)        Z_OBJPROP(*zval_p)
#define Z_OBJPROP_PP(zval_pp)        Z_OBJPROP(**zval_pp)

#define Z_OBJ_HANDLER(zval, hf)     Z_OBJ_HT((zval))->hf
#define Z_OBJ_HANDLER_P(zval_p, h)    Z_OBJ_HANDLER(*zval_p, h)
#define Z_OBJ_HANDLER_PP(zval_p, h)        Z_OBJ_HANDLER(**zval_p, h)

#define Z_OBJDEBUG(zval,is_tmp)        (Z_OBJ_HANDLER((zval),get_debug_info)?    \
                        Z_OBJ_HANDLER((zval),get_debug_info)(&(zval),&is_tmp TSRMLS_CC):
                        (is_tmp=0,Z_OBJ_HANDLER((zval),get_properties)?Z_OBJPROP(zval):NU
#define Z_OBJDEBUG_P(zval_p,is_tmp)    Z_OBJDEBUG(*zval_p,is_tmp)
#define Z_OBJDEBUG_PP(zval_pp,is_tmp)    Z_OBJDEBUG(**zval_pp,is_tmp)

//操作资源的
#define Z_RESVAL(zval)            (zval).value.lval
#define Z_RESVAL_P(zval_p)        Z_RESVAL(*zval_p)
#define Z_RESVAL_PP(zval_pp)        Z_RESVAL(**zval_pp)
```

links

- 目录
- 2.1 变量的类型
- 2.3 创建PHP变量

2.3 创建PHP变量

我们已经知道php变量在内核中其实是通过zval结构来实现的，也初步了解如何设置一个zval结构的类型和值。 这一节我们将在前两节的基础上，彻底掌握对zval结构的操控，其间将引入很多超棒的新宏。

在code的时候，很希望在内核中创建的zval可以让用户在PHP语言里以变量的形式使用，为了实现这个功能， 我们首先要创建一个zval。最容易想到的办法便是创建一个zval指针， 然后申请一块内存并让指针指向它。如果你脑海里浮现出了malloc(sizeof(zval))的影子， 那么请你立即刹车，不要用malloc来做这件事情，内核给我们提供了相应的宏来处理这件事， 理由和以前一样：为了代码漂亮并保持版本升级时的兼容性。 这个宏的是：MAKE_STD_ZVAL(pzv)。这个宏会用内核的方式来申请一块内存并将其地址付给pzv，并初始化它的refcount和is_ref两个属性，更棒的是，它不但会自动的处理内存不足问题， 还会在内存中选个最优的位置来申请。

> 除了MAKE_STD_ZVAL()宏函数，ALLOC_INIT_ZVAL()宏函数也是用来干这件事的， 唯一的不同便是它会将pzv所指的zval的类型设置为IS_NULL;

申请完空间后，我们便可以给这个zval赋值了。基于已经介绍的宏， 也许我们需要Z_TYPE_P(p) = IS_NULL来设置其是null类型，并用Z_SOMEVAL形式的宏来为它赋值， 但是现在你有了更好更短的选择！

内核中提供一些宏来简化我们的操作，可以只用一步便设置好zval的类型和值。

新宏	其它宏的实现方法
ZVAL_NULL(pvz); **(注意这个Z和VAL之间没有下划线！)**	Z_TYPE_P(pzv) = IS_NULL;**(IS_NULL型不用赋值，因为这个类型只有一个值就是null，^_^)**
ZVAL_BOOL(pzv, b); **(将pzv所指的zval设置为IS_BOOL类型，值是b)**	Z_TYPE_P(pzv) = IS_BOOL; Z_BVAL_P(pzv) = b ? 1 : 0;
ZVAL_TRUE(pzv); **(将pzv所指的zval设置为IS_BOOL类型，值是true)**	ZVAL_BOOL(pzv, 1);
ZVAL_FALSE(pzv); **(将pzv所指的zval设置为IS_BOOL类型，值是false)**	ZVAL_BOOL(pzv, 0);
ZVAL_LONG(pzv, l); **(将pzv所指的zval设置为IS_LONG类型，值是l)**	Z_TYPE_P(pzv) = IS_LONG; Z_LVAL_P(pzv) = l;
ZVAL_DOUBLE(pzv, d); **(将pzv所指的zval设置为IS_DOUBLE类型，值是d)**	Z_TYPE_P(pzv) = IS_DOUBLE; Z_DVAL_P(pzv) = d;
ZVAL_STRINGL(pzv,str,len,dup);**(下面单独解释)**	Z_TYPE_P(pzv) = IS_STRING; Z_STRLEN_P(pzv) = len; if (dup) {Z_STRVAL_P(pzv) =estrndup(str, len + 1);} else {Z_STRVAL_P(pzv) = str;}
ZVAL_STRING(pzv, str, dup);	ZVAL_STRINGL(pzv, str,strlen(str), dup);
ZVAL_RESOURCE(pzv, res);	Z_TYPE_P(pzv) = IS_RESOURCE; Z_RESVAL_P(pzv) = res;

ZVAL_STRINGL(pzv,str,len,dup)中的dup参数

先阐述一下ZVAL_STRINGL(pzv,str,len,dup); str和len两个参数很好理解，因为我们知道内核中保存了字符

串的地址和它的长度， 后面的dup的意思其实很简单，它指明了该字符串是否需要被复制。 值为 1 将先申请一块新内存并赋值该字符串，然后把新内存的地址复制给pzv， 为 0 时则是直接把str的地址赋值给zval。

《抚琴居》上的一篇文章说这项特性将会在你仅仅需要创建一个变量并将其指向一个已经由 Zend 内部数据内存时变得很有用。

ZVAL_STRINGL与ZVAL_STRING的区别

如果你想在某一位置截取该字符串或已经知道了这个字符串的长度， 那么可以使用宏ZVAL_STRINGL(zval, string, length, duplicate)， 它显式的指定字符串长度， 而不是使用strlen()。这个宏该字符串长度作为参数。但它是二进制安全的，而且速度也比ZVAL_STRING快，因为少了个strlen。

ZVAL_RESOURCE约等于ZVAL_LONG

上一节中我们说过PHP中的资源类型的值其实就是一个整数，所以ZVAL_RESOURCE和ZVAL_LONG的工作差不多， 只不过它会把zval的类型设置为 IS_RESOURCE。

links

- 目录
- 2.2 变量的值
- 2.4 变量的存储方式

2.4 变量的存储方式

我们在前两节已经了解了PHP中变量的类型和值是怎样在内核中用C语言实现的， 这一节我们将看一下内核是怎样来组织用户在PHP中定义的变量的。

有一点对我们扩展开发者来说非常棒，那就是用户在PHP中定义的变量我们都可以在一个HashTable中找到， 当PHP中定义了一个变量，内核会自动的把它的信息储存到一个用HashTable实现的符号表里。

全局作用域的符号表是在调用扩展的RINIT方法(一般都是MINIT方法里)前创建的，并在RSHUTDOWN方法执行后自动销毁。

当用户在PHP中调用一个函数或者类的方法时，内核会创建一个新的符号表并激活之， 这也就是为什么我们无法在函数中使用在函数外定义的变量的原因 （因为它们分属两个符号表，一个当前作用域的，一个全局作用域的）。 如果不是在一个函数里，则全局作用域的符号表处于激活状态。

我们现在打开Zend/zend_globals.h文件，看一下_zend_execution_globals结构体，会在其中发现这么两个element：

```
struct _zend_executor_globals {
    ...
    HashTable symbol_table;
    HashTable *active_symbol_table;
    ...
};
```

其中的 symbol_table元素可以通过EG宏来访问，它代表着PHP的全局变量，如$GLOBALS，其实从根本上来讲， $GLOBALS不过是EG(symbol_table)的一层封装而已。

与之对应，下面的active_symbol_table元素也可以通过EG(active_symbol_table)的方法来访问，它代表的是处于当前作用域的变量符号表。

我们上边也看到了，其实这两个成员在_zend_executor_globals里虽然都代表HashTable， 但一个是真正的HashTable，而另一个是一个指针。 当我们在对HashTable进行操作的时候，往往是把它的地址传递给一些函数。 所以，如果我们要对EG(symbol_table)的结果进行操作，往往需要对它进行求址操作然后用它的地址作为被调用函数的参数。

下面我们用一段例子来解释下上面说的理论：

```
<?php
$foo = 'bar';
?>
```

上面是一段PHP语言的例子，我们创建了一个变量，并把它的值设置为'bar'，在以后的代码中我们便可以使用$foo变量。相同的功能我们怎样在内核中实现呢？我们可以先构思一下步骤：

- 创建一个zval结构，并设置其类型。
- 设置值为'bar'。

- 将其加入当前作用域的符号表，只有这样用户才能在PHP里使用这个变量。
- 具体的代码为：

```
{
    zval *fooval;

    MAKE_STD_ZVAL(fooval);
    ZVAL_STRING(fooval, "bar", 1);
    ZEND_SET_SYMBOL( EG(active_symbol_table) ,  "foo" , fooval);
}
```

首先，我们声明一个zval指针，并申请一块内存。然后通过ZVAL_STRING宏将值设置为‘bar’,最后一行的作用就是将这个zval加入到当前的符号表里去，并将其label定义成foo，这样用户就可以在代码里通过$foo来使用它了。

links

- 目录
- 2.3 创建PHP变量
- 2.5 变量的检索

2.5 变量的检索

用户在PHP语言里定义的变量，我们能否在内核中获取到呢？答案当然是肯定的，下面我们就看如何通过zend_hash_find()函数来找到当前某个作用域下用户已经定义好的变量。 zend_hash_find()函数是内核提供的操作HashTable的API之一，如果你没有接触过，可以先记住怎么使用就可以了。

```
{
    zval **fooval;

    if (zend_hash_find(
            &EG(active_symbol_table), //这个参数是地址，如果我们操作全局作用域，则需要&EG(symbol_
            "foo",
            sizeof("foo"),
            (void**)&fooval
        ) == SUCCESS
    )
    {
        php_printf("成功发现$foo!");
    }
    else
    {
        php_printf("当前作用域下无法发现$foo.");
    }
}
```

首先我们定义了一个指向指针的指针，然后通过zend_hash_find去EG(active_symbol_table)作用域下寻找名称为foo($foo)的变量， 如果成功找到，此函数将返回SUCCESS。看完代码，你肯定有很多疑问。 为什么还要进行`sizeof("foo")`运算，fooval明明是`zval**`型的，为什么转成`void**`的？而且为什么还要进行&fooval运算，fooval本身不就已经是指向指针的指针了吗？:-)， 该回答的问题确实很多，不要过于担心，让我们带着这些问题继续往下走。

首先要说明的是，内核定义HashTable这个结构，并不是单单用来储存PHP语言里的变量的， 其它很多地方都在应用HashTable(**这就是个神器)**。 一个HashTable有很多元素，在内核里叫做bucket。然而每个bucket的大小是固定的， 所以如果我们想在bucket里存储任意数据时，最好的办法便是申请一块内存保存数据， 然后在bucket里保存它的指针。以zval *foo*为例， 内核会先申请一块足够保存指针内存来保存*foo*，比如这块内存的地址是*p*，也就是*p=&foo*， 并在*bucket*里保存*p*，这时我们便明白了，*p*其实就是*`zval**`*类型的。 至于*bucket*为什么保存*`zval**`*类型的指针，而不是直接保存`*zval*`类型的指针，我们到下一章在详细叙述。

所以当我们去HashTable里寻找变量的时候，得到的值其实是一个zval的指针。 In order to populate that pointer into a calling function's local storage, the calling function will naturally dereference the local pointer, resulting in a variable of indeterminate type with two levels of indirection (such as `void**`). Knowing that your "indeterminate type" in this case is `zval*`, you can see where the type being passed into zend_hash_find() will look different to the compiler, having three levels of indirection rather than two. This is done on purpose here so a simple typecast is added to the function call to silence compiler warnings.

如果zend_hash_find()函数找到了我们需要的数据，它将返回SUCCESS常量， 并把它的地址赋给我们在调用zend_hash_find()函数传递的fooval参数， 也就是说此时fooval就指向了我们要找的数据。如果没有找到，那它不会对我们fooval参数做任何修改，并返回FAILURE常量。

就去符号表里找变量而言，SUCCESS和FAILURE仅代表这个变量是否存在而已。

links

- 目录
- 2.4 变量的存储方式
- 2.6 类型转换

2.6 类型转换

现在我们已经可以从符号表中获取用户在PHP语言里定义的变量了，是该做点其它事的时候了，举个例子，比如给它来个类型转换:-)。想想C语言中的类型转换细则，你的头是不是已经大了？但是变量的类型转换就是如此重要，如果没有，那我们的代码就会是下面这样了:

```
void display_zval(zval *value)
{
    switch (Z_TYPE_P(value)) {
        case IS_NULL:
            /* 如果是NULL，则不输出任何东西 */
            break;

        case IS_BOOL:
            /* 如果是bool类型，并且true，则输出1，否则什么也不干 */
            if (Z_BVAL_P(value)) {
                php_printf("1");
            }
            break;
        case IS_LONG:
            /* 如果是long整型，则输出数字形式 */
            php_printf("%ld", Z_LVAL_P(value));
            break;
        case IS_DOUBLE:
            /* 如果是double型，则输出浮点数 */
            php_printf("%f", Z_DVAL_P(value));
            break;
        case IS_STRING:
            /* 如果是string型，则二进制安全的输出这个字符串 */
            PHPWRITE(Z_STRVAL_P(value), Z_STRLEN_P(value));
            break;
        case IS_RESOURCE:
            /* 如果是资源，则输出Resource #10 格式的东东 */
            php_printf("Resource #%ld", Z_RESVAL_P(value));
            break;
        case IS_ARRAY:
            /* 如果是Array，则输出Array5个字母！ */
            php_printf("Array");
            break;
        case IS_OBJECT:
            php_printf("Object");
            break;
        default:
            /* Should never happen in practice,
             * but it's dangerous to make assumptions
             */
             php_printf("Unknown");
             break;
    }
}
```

看完上面的代码，你是不是有点似曾相识的感觉？o(∩∩)o...哈哈，和直接<?php echo $foo;?>这个简单到极点的php语句来比，上面的实现算是天书了。当然，真正的环境并没有这么囧，内核中提供了好多函数专

门来帮我们实现类型转换的功能，你需要的只是调用一个函数而已。这一类函数有一个统一的形式：convert*to**()

```
//将任意类型的zval转换成字符串
void change_zval_to_string(zval *value)
{
    convert_to_string(value);
}

//其它基本的类型转换函数
ZEND_API void convert_to_long(zval *op);
ZEND_API void convert_to_double(zval *op);
ZEND_API void convert_to_null(zval *op);
ZEND_API void convert_to_boolean(zval *op);
ZEND_API void convert_to_array(zval *op);
ZEND_API void convert_to_object(zval *op);

ZEND_API void _convert_to_string(zval *op ZEND_FILE_LINE_DC);
#define convert_to_string(op) if ((op)->type != IS_STRING) { _convert_to_string((op) ZEND
```

这里面有两个比较特殊，一个就是convert_to_string其实是一个宏函数，调用的另外一个函数；第二个便是没有convert_to_resource()的转换函数，因为资源的值在用户层面上，根本就没有意义，内核不会对它的值(不是指那个数字)进行转换。

好了，我们用php的echo的时候会先把变量转换成字符串，但是我们看见convert*to_string*的参数是*zval**的，你是不是开始担心在进行数据转换时破坏了原来数据的值？而我们 `<?php $a=intval($b);` 并不会破坏*$b*的值。把原来的值破坏掉的做法绝对不是一个好主意，内核中在*echo*一个变量的时候也不是这样做的。在下一章，我们将知道怎样便可以在不损坏原变量值的情况下，进行*convert_to*类操作。

links

- 目录
- 2.5 变量的检索
- 2.7 第二章小结

2.7 小结

在这一章我们了解了php变量在内核中是如何实现的，我们已经可以识别出一个变量的类型，把它加到符号表去或者从符号表中找出等等等等。在下一章我们的目光开始转向内存，顺道研究下怎样复制已经存在的zval，以及如何在它们没用的时候及时的清理掉，还有最重要的，怎么不使用copy，而使用引用！

我们已经了解到zend引擎中针对一个请求的内存管理层，了解了常驻内存与非常驻内存的概念与区别。在读完下一章后，我们便有了比较完整的理论基础来在我们自己的扩展中灵活的操作各个变量。

links

- 目录
- 2.6 类型转换
- 3 内存管理

3 内存管理

脚本语言与编译型语言最根本的区别可能就在内存管理上。但这并不限于脚本语言，现在越来越多的语言不再允许用户直接操作内存，而由虚拟机来代替用户负责内存的分配及回收，如C#、Java、PHP等。

links

- 2.7 第二章小结
- 3.1 内存管理

3.1 内存管理

在PHP里，我们可以定义字符串变量，比如<?php $str="hello";?>，$str这个字符串变量可以被自由的修改与复制等。这一切在C语言里看起来都是不可能的事情，我们用#char *p = "hello";#来定义一个字符串，但它是常量，是不能被修改的，如果你用p[1]='c';来修改这个字符串会引发段错误(Gcc,c99)，为了修改C语言里的字符串常量，我们往往需要定义字符串数组。为了得到一个能够让我们自由修改的字符串，我们往往需要用strdup函数来复制一个字符串出来。

```
{
    char *p = "hello world";
    // p[0] = 'a'; 如果这么做，就等着运行时段错误吧。
    char *str;
    str = strdup(p);
    str[0] = 'a'; //这时就能自由修改了。
}
```

在PHP内核中，大多数情况下都不应该直接使用C语言中自带着malloc、free、strdup、realloc、calloc等操作内存的函数，而应使用内核提供的操作内存的函数，这样可以由内核整体统一的来管理内存。

Free the Mallocs

每个平台操作内存的方式都是差不多的有两个方面，一负责申请，二负责释放。如果应用程序向系统申请内存，系统便会在内存中寻找还没有被使用的地方，如果有合适的，便分配给这个程序，并标记下来，不再给其它的程序了。如果一个内存块没有释放，而所有者应用程序也永远不再使用它了。那么，我们就称其为"内存泄漏"，那么这部分内存就无法再为其它程序所用了。 在一个典型的客户端应用程序中，偶尔的小量的内存泄漏是可以被操作系统容忍的，因为在进程结束后该泄漏内存会被返回给OS。这并没有什么高科技含量，因为OS知道它把该内存分配给了哪个程序，并且它能够在一个程序结束后把这些内存给回收回来。 但是，世界总是不缺乏特例！对于一些需要长时间运行的程序，比如像Apache这样的web服务器以及它的php模块来说，都是伴随着操作系统长时间运行的，所以OS在很长一段时间内不能主动的回收内存，从而导致这个程序的每一个内存泄漏都会促进量变到质变的进化，最终引起严重的内存泄漏错误，使系统的资源消耗殆尽。现在，我们来在C语言中故意错误的模拟一下PHP的stristr()函数为例，为了使用大小写不敏感的方式来搜索一个字符串，我们需要创建两个辅助的字符串，它们分别是被查找字符串和待查找字符串的小写化副本，然后由这两个副本来帮助我们来完成这次搜索。如果我们在执行这个函数后不释放这些副本占用的资源，那么每一次stristr函数都将是对内存的一次永远的侵占，最终导致这个函数占用了所有的系统内存，而没有实际意义！ 大多数人提出来的理想的解决方案是：书写优秀，整洁并且风格一致的代码，这当然是毫无疑问的。但是在PHP扩展开发这样的底层环境中，这并不能解决全部的问题。比如，你需要自己保证在层层嵌套调用中对某块内存的使用都是正确的，且会及时释放的。[1]

错误处理

为了实现从用户端(PHP语言中)"跳出"，需要使用一种方法来完全"跳出"一个活动请求。这个功能是在内核中实现的：在一个请求的开始设置一个"跳出"地址，然后在任何die()或exit()调用或在遇到任何关键错误(E_ERROR)时执行一个longjmp()以跳转到该"跳出"地址。

```
void call_function(const char *fname, int fname_len TSRMLS_DC)
```

```
{
    zend_function *fe;
    char *lcase_fname;
    /* php函数的名字是大小写不敏感的
     * 我们可以在function tables里找到他们
     * 保存的所有函数名都是小写的。
     */
    lcase_fname = estrndup(fname, fname_len);
    zend_str_tolower(lcase_fname, fname_len);

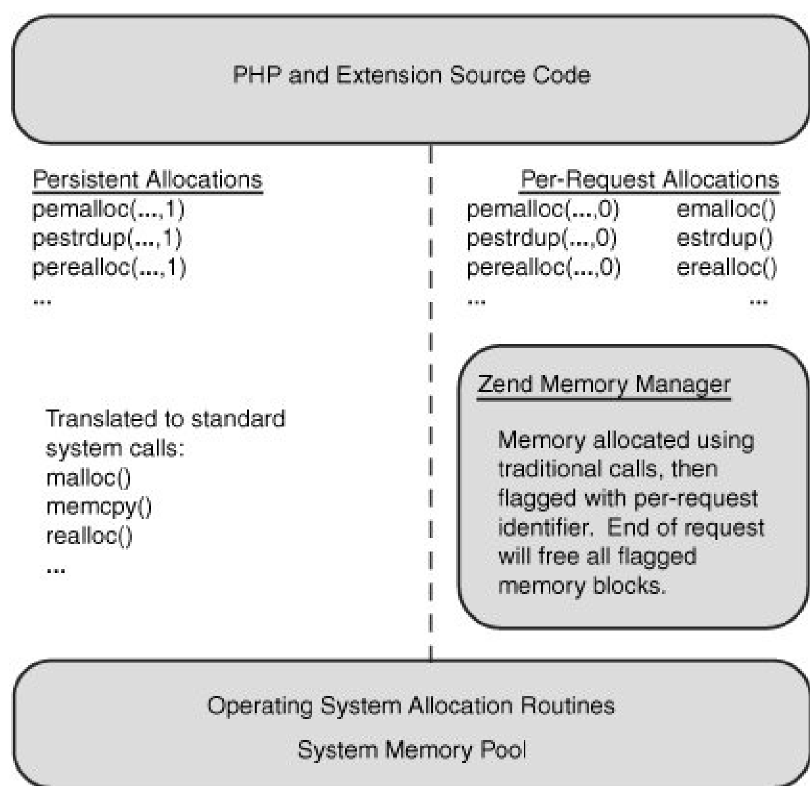
    if (zend_hash_find(EG(function_table),lcase_fname, fname_len + 1, (void **)&fe) == SU
    {
        zend_execute(fe->op_array TSRMLS_CC);
    }
    else
    {
        php_error_docref(NULL TSRMLS_CC, E_ERROR,"Call to undefined function: %s()", fnam
    }
    efree(lcase_fname);
}
```

当php*error_docref*这个函数被调用的时候，便会触发内核中的错误处理机制，根据错误级别来决定是否调用*longjmp*来终止当前请求并退出*call_function*函数，从而*efree*函数便永远不会被执行了。

其实*php_error_docref()*函数就相当于*php*语言里的*trigger_error()*函数.它的第一个参数是一个将被添加到*docref*的可选的文档引用第三个参数可以是任何我们熟悉的*E*
*家族常量，用于指示错误的严重程度。后面的两个参数就像printf()风格的格式化和变量参数列表式样。

Zend内存管理器

在上面的"跳出"请求期间解决内存泄漏的方案之一是：使用Zend内存管理(Zend Memory Manager,简称ZendMM、ZMM)层。内核的这一部分非常类似于操作系统的内存管理功能——分配内存给调用程序。区别在于，它处于进程空间中非常低的位置而且是"请求感知"的；这样一来，当一个请求结束时，它能够执行与OS在一个进程终止时相同的行为。也就是说，它会隐式地释放所有的为该请求所占用的内存。图1展示了ZendMM与OS以及PHP进程之间的关系。



除了提供隐式的内存清除功能之外，ZendMM还能够根据php.ini中memory_limit设置来控制每一次内存请求行为，如果一个脚本试图请求比系统中可用内存更多的内存，或大于它每次应该请求的最大量，那么，ZendMM将自动地发出一个E_ERROR消息并且启动相应的终止进程。这种方法的一个额外优点在于，大多数内存分配调用的返回值并不需要检查，因为如果失败的话将会导致立即跳转到引擎的退出部分。 把PHP内核代码和OS的实际的内存管理层"钩"在一起的原理并不复杂：所有内部分配的内存都要使用一组特定的可选函数实现。例如，PHP内核代码不是使用malloc(16)来分配一个16字节内存块而是使用了emalloc(16)。除了实现实际的内存分配任务外，ZendMM还会使用相应的绑定请求类型来标志该内存块；这样以来，当一个请求"跳出"时，ZendMM可以隐式地释放它。 有些时候，某次申请的内存需要在一个请求结束后仍然存活一段时间，也就是持续性存在于各个请求之间。这种类型的分配（因其在一次请求结束之后仍然存在而被称为"永久性分配"），可以使用传统型内存分配器来实现，因为这些分配并不会添加ZendMM使用的那些额外的相应于每种请求的信息。然而有时，我们必须在程序运行时根据某个数据的具体值或者状态才能确定是否需要进行永久性分配，因此ZendMM定义了一组帮助宏，其行为类似于其它的内存分配函数，但是使用最后一个额外参数来指示是否为永久性分配。 如果你确实想实现一个永久性分配，那么这个参数应该被设置为1；在这种情况下，请求是通过传统型malloc()分配器家族进行传递的。然而，如果运行时刻逻辑认为这个块不需要永久性分配；那么，这个参数可以被设置为零，并且调用将会被调整到针对每种请求的内存分配器函数。 例如，pemalloc(buffer_len，1)将映射到malloc(buffer_len)，而pemalloc(buffer_len，0)将被使用下列语句映射到emalloc(buffer_len)：

```
//define in Zend/zend_alloc.h:
#define pemalloc(size, persistent)    ((persistent)?malloc(size): emalloc(size))
```

所有这些在ZendMM中提供的内存管理函数都能够从下表中找到其在C语言中的函数。

C语言原生函数	PHP内核封装后的函数
void *malloc(size_t count);*	void emalloc(size_t count); void *pemalloc(size_t count, char persistent);*
	void *ecalloc(size_t count);*

	void pecalloc(size_t count, char persistent);
void *realloc(void* ptr, size_t count);	void *erealloc(void* ptr, size_t count); void *perealloc(void* ptr, size_t count, char persistent);
void *strdup(void* ptr);	void *estrdup(void* ptr); void *pestrdup(void* ptr, char persistent);
void free(void *ptr);*	void efree(void ptr); void pefree(void *ptr, char persistent);

你可能会注意到，即使是pefree()函数也要求使用永久性标志。这是因为在调用pefree()时，它实际上并不知道是否ptr是一种永久性分配。需要注意的是，如果针对一个ZendMM申请的非永久性内存直接调用free()能够导致双倍的空间释放，而针对一种永久性分配调用efree()有可能会导致一个段错误，因为ZendMM需要去查找并不存在的管理信息。因此，你的代码需要记住它申请的内存是否是永久性的，从而选择不同的内存函数，free()或者efree()。 除了上述内存管理函数外，还存在其它一些非常方便的ZendMM函数，例如：

```
void *estrndup(void *ptr, int len);
```

该函数能够分配len+1个字节的内存并且从ptr处复制len个字节到最新分配的块。这个estrndup()函数的行为可以大致描述如下：

```
ZEND_API char *_estrndup(const char *s, uint length ZEND_FILE_LINE_DC ZEND_FILE_LINE_ORIG
{
    char *p;

    p = (char *) _emalloc(length+1 ZEND_FILE_LINE_RELAY_CC ZEND_FILE_LINE_ORIG_RELAY_CC);
    if (UNEXPECTED(p == NULL))
    {
        return p;
    }
    memcpy(p, s, length);
    p[length] = 0;
    return p;
}
```

在此，被隐式放置在缓冲区最后的0可以确保任何使用estrndup()实现字符串复制操作的函数都不需要担心会把结果缓冲区传递给一个例如printf()这样的希望以为NULL为结束符的函数。当使用estrndup()来复制非字符串数据时，最后一个字节实质上浪费了，但其中的利明显大于弊。

```
void *safe_emalloc(size_t size, size_t count, size_t addtl);
void *safe_pemalloc(size_t size, size_t count, size_t addtl, char persistent);
```

这些函数分配的内存空间最终大小都是((size*count)+addtl)。 你可以会问："为什么还要提供额外函数呢？为什么不使用一个emalloc/pemalloc呢？"。 原因很简单：为了安全，以防万一。尽管有时候可能性相当小，但是，正是这一"可能性相当小"的结果导致宿主平台的内存溢出。 这可能会导致分配负数个数的字节空间，或更有甚者，会导致分配一个小于调用程序要求大小的字节空间。 而safe_emalloc()能够避免这种类型的陷井-通过检查整数溢出并且在发生这样的溢出时显式地预以结束。

> 注意，并不是所有的内存分配例程都有一个相应的p*对等实现。例如，不存在pestrndup()，并且在PHP 5.1版本前也不存在safe_pemalloc()。

贡献者名单
　绊大象的蚂蚁{eaglevean#163.com}，注1

links

- 3 内存管理
- 3.2 引用计数

3.2 内存管理

对于PHP这种需要同时处理多个请求的程序来说，申请和释放内存的时候应该慎之又慎，一不小心便会酿成大错。另一方面，除了要安全的申请和释放内存外，还应该做到内存的最小化使用，因为它可能要处理每秒钟数以千计的请求，为了提高系统整体的性能，每一次操作都应该只使用最少的内存，对于不必要的相同数据的复制则应该能免则免。我们来看下面这段PHP代码：

```php
<?php
$a = 'Hello World';
$b = $a;
unset($a);
```

第一条语句执行后，PHP创建了$a这个变量，并为它申请了12B的内存来存放"hello world"这个字符串（最后加个NULL字符，你懂的）。紧接着把$a赋给了$b，并释放掉$a; 对于PHP来说，如果每一次变量赋值都执行一次内存复制的话，那需要额外申请12B的内存来存放这个重复的数据,当然为了复制内存，还需要cpu执行某些计算，这当然会加重cpu的负载。当第三句执行后，$a被释放了，我们刚才的设想突然变的这么滑稽，这次赋值显得好多余哦。如果早就知道$a不用了，那我们直接让$b用$a的内存不就行了，还赋值干嘛？如果你觉得12B没什么，那设想下如果$a是个10M的文件内容，或者20M，是不是我们的计算机资源消耗的有点冤枉呢？别担心，PHP很聪明！前面章节说过，PHP变量的名称和值在内核中是保存在两个不同的地方的，值是通过一个与名字毫无关系的zval结构来保存，而这个变量的名字a则保存在符号表里，两者之间通过指针联系着。在我们上面的例子里，$a是一个字符串，我们通过zend_hash_add把它添加到符号表里，然后又把它赋值给$b,两者拥有相同的内容！如果两者指向完全相同的内容，我们有什么优化措施吗？

```c
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),&helloval, sizeof(zval*), NUL
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),&helloval, sizeof(zval*), NUL
    //通过这个例子我们看出了，我们可以把$a和$b都指向helloval~!
```

现在我们检查$a和$b两个变量，他们的值指向了"hello world"这个字符串在内存中的位置。但是在第三行：unset($a);这条语句释放了$a。在这种情况下，unset函数并不知道$a的值同时被$b用着，所以如果它直接释放内存，则会导致$b的值也被清空了，从而导致逻辑错误，甚至可能会导致系统崩溃。呵呵，其实你心里明白，PHP不会让上述问题发生的！回顾一下zval的四个成员value、type、is_ref**gc**、**refcount**gc，我们对value和type已经很熟了，现在则是后两个成员发挥威力的时候了，这里我们主要讲解refcount**gc**这个成员。当一个变量被第一次创建的时候，它对应的**zval**结构体的**refcount**gc成员的值会被初始化为1，理由很简单，因为只有这个变量自己在用它。但是当你把这个变量赋值给别的变量时，refcount__gc属性便会加1变成2，因为现在有两个变量在用这个zval结构了！以上描述转为内核中的代码大体如下：

```c
    zval *helloval;
    MAKE_STD_ZVAL(helloval);
    ZVAL_STRING(helloval, "Hello World", 1);
    zend_hash_add(EG(active_symbol_table), "a", sizeof("a"),&helloval, sizeof(zval*), NUL
    ZVAL_ADDREF(helloval); //这句很特殊，我们显式的增加了helloval结构体的refcount
```

```
    zend_hash_add(EG(active_symbol_table), "b", sizeof("b"),&helloval, sizeof(zval*), NUL
```

这个时候当我们再用unset删除$a的时候，它删除符号表里的$a的信息，然后清理它的值部分，这时它发现$a的值对应的zval结构的refcount值是2，也就是有另外一个变量在一起用着这个zval，所以unset只需把这个zval的refcount减去1就行了！

写时复制机制

引用计数绝对是节省内存的一个超棒的模式！但是当我们修改$b的值，而且还需要继续使用$a时，该怎么办呢？

```
    $a = 1;
    $b = $a;
    $b += 5;
```

从代码逻辑来看，我们希望语句执行后$a仍然是1，而$b则需要变成6。我们知道在第二句完成后内核通过让$a和$b共享一个zval结构来达到节省内存的目的，但是现在第三句来了，这时$b的改变应该怎样在内核中实现呢？答案非常简单，内核首先查看refcount__gc属性，如果它大于1则为这个变化的变量从原zval结构中复制出一份新的专属与$b的zval来，并改变其值。

```
zval *get_var_and_separate(char *varname, int varname_len TSRMLS_DC)
{
    zval **varval, *varcopy;
    if (zend_hash_find(EG(active_symbol_table),varname, varname_len + 1, (void**)&varval)
    {
        /* 如果在符号表里找不到这个变量则直接return */
        return NULL;
    }

    if ((*varval)->refcount < 2)
    {
        //如果这个变量的zval部分的refcount小于2，代表没有别的变量在用，return
        return *varval;
    }

    /* 否则，复制一份zval*的值 */
    MAKE_STD_ZVAL(varcopy);
    varcopy = *varval;

    /* 复制任何在zval*内已分配的结构*/
    zval_copy_ctor(varcopy);

    /* 从符号表中删除原来的变量
     * 这将减少该过程中varval的refcount的值
     */
    zend_hash_del(EG(active_symbol_table), varname, varname_len + 1);

    /* 初始化新的zval的refcount，并在符号表中重新添加此变量信息，并将其值与我们的新zval相关联。*/
    varcopy->refcount = 1;
    varcopy->is_ref = 0;
    zend_hash_add(EG(active_symbol_table), varname, varname_len + 1,&varcopy, sizeof(zval
```

```
    /* 返回新zval的地址 */
    return varcopy;
}
```

现在$b变量拥有了自己的zval，并且可以自由的修改它的值了。

Change on Write

如果用户在PHP脚本中显式的让一个变量引用另一个变量时，我们的内核是如何处理的呢？

```
    $a = 1;
    $b = &$a;
    $b += 5;
```

作为一个标准的PHP程序猿，我们都知道$a的值也变成6了。当我们更改$b的值时，内核发现$b是$a的一个用户端引用，也就是所它可以直接改变$b对应的zval的值，而无需再为它生成一个新的不同与$a的zval。因为他知道$a和$b都想得到这次变化！ 但是内核是怎么知道这一切的呢？简单的讲，它是通过zval的is_ref**gc**成员来获取这些信息的。这个成员只有两个值，就像开关的开与关一样。它的这两个状态代表着它是否是一个用户在**PHP**语言中定义的引用。在第一条语句**($a = 1;)**执行完毕后**,$a**对应的**zval**的**refcount**gc等于1，is_ref**gc**等于**0;**。 当第二条语句执行后**($b = &$a;)**，**refcount**gc属性向往常一样增长为2，而且is_ref__gc属性也同时变为了1！ 最后，在执行第三条语句的时候，内核再次检查$b的zval以确定是否需要复制出一份新的zval结构来，这次不需要复制，因为我们刚才上面的get_var_and_separate函数其实是个简化版，并且少写了一个条件：

```
/* 如果这个zval在php语言中是通过引用的形式存在的，或者它的refcount小于2，则不需要复制。*/
if ((*varval)->is_ref || (*varval)->refcount < 2) {
    return *varval;
}
```

这一次，尽管它的refcount等于2，但是因为它的is_ref等于1，所以也不会被复制。内核会直接的修改这个zval的值。

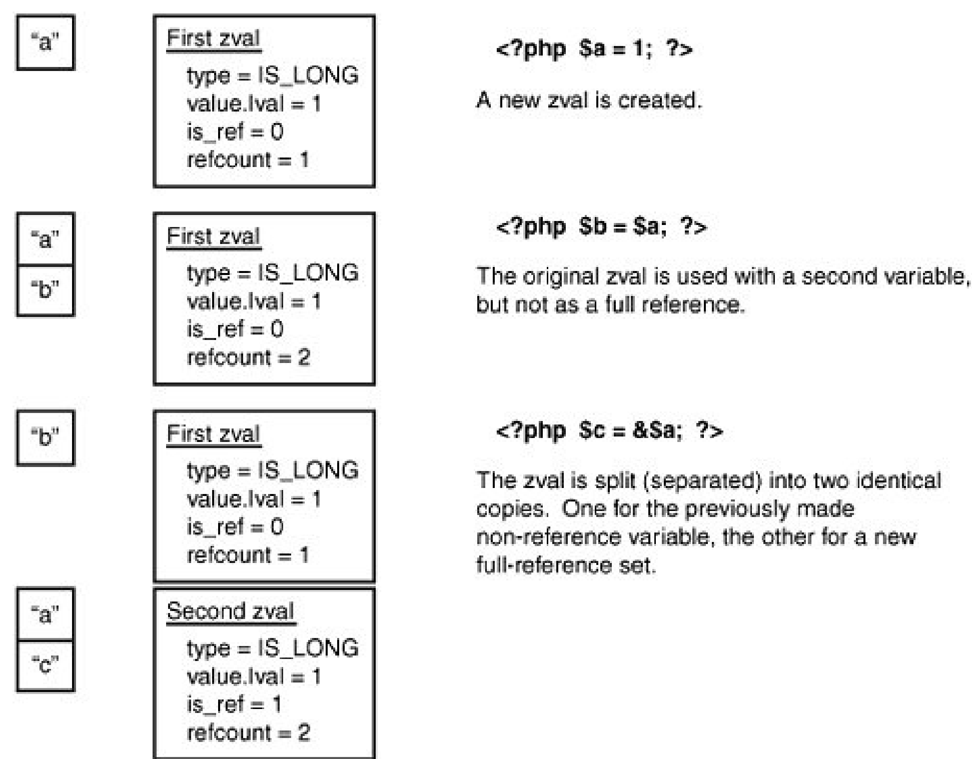
Separation Anxiety

我们已经了解了php语言中变量的复制和引用的一些事，但是如果复制和引用这两个事件被组合起来使用了该怎么办呢？看下面这段代码：

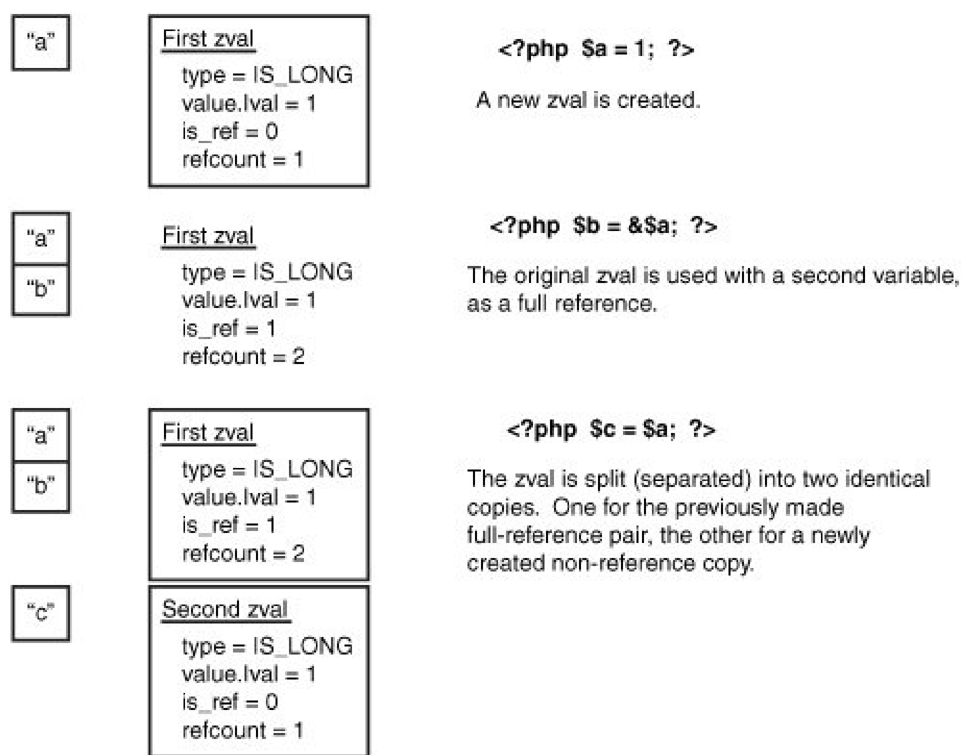
```
    $a = 1;
    $b = $a;
    $c = &$a;
```

这里我们可以看到,$a,$b,$c这三个变量现在共用一个zval结构，有两个属于change-on-write组合($a,$c),有两个属于copy-on-write组合($a,$b),我们的is_ref**gc**和**refcount**gc该怎样工作，才能正确的处理好这段复杂

的关系呢？ The answer is: 不可能！在这种情况下，变量的值必须分离成两份完全独立的存在！$a与$c共用一个zval,$b自己用一个zval，尽管他们拥有同样的值，但是必须至少通过两个zval来实现。见图3.2【在引用时强制复制！】



同样，下面的这段代码同样会在内核中产生歧义，所以需要强制复制！




```
//上图对应的代码
$a = 1;
$b = &$a;
$c = $a;
```

引用计数

需要注意的是，在这两种情况下，$b都与原初的zval相关联，因为当复制发生时，内核还不知道第三个变量的名字。

links

- 3.1 内存管理
- 3.3 3.3 第三章总结

3.3 内存管理

PHP是一种解释型的语言，对于用户而言，我们精心的控制内存意味着easier prototyping和更少的崩溃！当我们深入到内核之后，所有的安全防线都已经被越过，最终还是要依赖于真正有责任心的软件工程师来保证系统的稳定运行。

links

- 3.2 引用计数
- 4 动手编译PHP

4 动手编译PHP

到现在为止，你肯定应该在至少一种平台上安装过PHP，并用它来开发你的web程序了。你可能下载的win32平台下的iis或者apache对应的安装包，也可能使用了由第三方提供的linux、bsd等平台下的二进制包。而现在，则是我们动手自己编译PHP的时候了。这也是我们动手开发第一个扩展的最后一项准备知识了。 强烈推荐你在Linux下调试本章的程序，因为win部分我还没有翻译，:-)

links

- 3.3 3.3 第三章总结
- 4.1 编译前的准备

4.1 动手编译PHP

从一个PHP程序猿，到一个想为PHP开发扩展的程序猿，此间的进化有一步是跳不过去的，那就是你必须熟知如何编译PHP的源码。

*nix Tools

C语言的编译器是我们使用C语言的必备工具，你的系统应该已经自带了一种C语言的编译器，而且它极有可能是大名鼎鼎的GCC。通过检测你本机gcc或者cc程序的版本，可以很方便的知道你机器上是否已经安装的某种C语言的编译器。

```
walu@walu-ThinkPad-Edge:~$ gcc --version
gcc (Ubuntu/Linaro 4.5.2-8ubuntu4) 4.5.2
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

如果你还没有安装编译器，那你需要安装一个。最简单的办法便是去下载一个与你系统相符的rpm或者deb包，当然你也可以通过以下命令的一种来方便的安装：yum install gcc, apt-get install gcc, pkg-add -r gcc, 或者 emerge gcc. 除了编译器，你还需要以下程序：make, autoconf, automake, 和libtool。说实话，我连autoconf现在是啥还不知道(截至到现在，2011年9月6号)，不过除非RP太低，一般系统中都会自备了，而且phpize程序会把这些需要的脚本给生成好的。 对于编译需要的程序以及它们的版本我们可以在PHP官网找到最新的答案：

- autoconf: 2.13 (2.59+ for PHP 5.4+)
- automake: 1.4+
- libtool: 1.4.x+ (except 1.4.2)
- bison: 1.28, 1.35, 1.75, 2.0 or higher
- flex (PHP 5.2 and earlier): 2.5.4 (not higher)
- re2c: 0.13.4+

你千万不要被上面的清单给吓着，其实系统应该给装备好了，除非真RP低，那你出门去买张彩票吧... ... 当然，我们也可以通过SVN从PHP源码库里导出一份源码，需要注意的是，PHP的svn源码库地址是https协议的。官方推荐我们直接签出它的php-src目录:

```
$ svn checkout https://svn.php.net/repository/php/php-src --depth immediates php-src
$ cd php-src
```

当然，我们也可以签出特定的版本：

- PHP 5.3: svn checkout https://svn.php.net/repository/php/php-src/branches/PHP_5_3 php-src-5.3
- PHP 5.4: svn checkout https://svn.php.net/repository/php/php-src/branches/PHP_5_4 php-src-5.4
- PHP HEAD: svn checkout https://svn.php.net/repository/php/php-src/trunk php-src-trunk

最新的大家可以来这查看：http://php.net/svn.php

Win32 Tools

这里仅代表作者05年的观点，我还没有在win平台下测试过，稍後会把这段修正过来。

```
The Win32/PHP5 build system is a complete rewrite and represents a significant leap forwa
```

First, you'll need to grab libraries and development headers used by many of the core PHP extensions. Fortunately, many of these files are redistributed from php.net as a single .zip file located at http://www.php.net/extra/win32build.zip.

Create a new directory named C:\PHPDEV\ and unzip win32build.zip using your favorite zip management program into this location. The folder structure contained in the zip file will create a subdirectory, C:\PHPDEV\win32build, which will contain further subfolders and files. It's not necessary to name your root folder PHPDEV; the only important thing is that win32build and the PHP source tree are both children of the same parent folder.

Next you'll need a compiler. If you've already got Visual C++ .NET you have what you need; otherwise, download Visual C++ Express from Microsoft at http://lab.msdn.microsoft.com/express/.

The installer, once you've downloaded and run it, will display the usual welcome, EULA (End-User License Agreement), and identification dialogs. Read through these screens and proceed using the Next buttons after you've agreed to the terms of the EULA and entered any appropriate information.

Installation location is of course up to you, and a typical installation will work just fine. If you'd like to create a leaner installation, you may deselect the three optional componentsGUI, MSDN, and SQL Server.

The final package is the Platform SDK, also available for download from Microsoft at http://www.microsoft.com/downloads/details.aspx?FamilyId=A55B6B43-E24F-4EA3-A93E-40C0EC4F68E5. The site currently lists three download options: PSDK-x86.exe, PSDK-ia64.exe, and PSDK-amd64.exe. These options refer to x86 compatible 32bit, Intel64bit, and AMD64bit processors respectively. If you're not sure which one applies to your processor, select PSDK-x86.exe, which should work cleanly, albeit less efficiently, with both 64 bit variants.

As before, proceed through the first few screens as you would with any other installer package until you are prompted to select between Typical and Custom installation. A Typical installation includes the Core SDK package, which is sufficient for the purposes of building PHP. Other packages can be deselected by choosing a Custom installation, but if you have the hard disk space to spare, you might as well install it all. The other packages may come in handy later on.

So unless you're byte conscious, select Typical and proceed through the next couple of standard issue screens until the installer begins copying and registering files. This process should take a few minutes so grab some popcorn.

Once installation is complete you'll have a new item on your Start menuMicrosoft Platform SDK for

Windows Server 2003 SP1.

获取PHP源码

其实你有很多办法安装PHP，最简单的一种就是从你系统的库或者源里通过apt-get、yum install之类的命令直接安装PHP5，这样做的好处你的系统可能会自动处理一些php在它上面的工作时的一些bug，而且你还可以方便的升级与卸载。这样做也有缺点，那就是你的PHP版本永远无法是最新的，通常www.php.net发布数周甚至数月后你才能用上相应的版本。 第二种方法：也是推荐使用的一种方法，那就是自行下载php-x.y.z.tar.gz的源码包，然后自行编译安装。这种包一般都是经过了海量的测试后才发布的，而且非常接近最新beta或者alpha版本。 此外，你还可以snaps.php.net提供的快照包来下载php进行编译安装，这个站点每几个小时便会从源码库里打包出一份新的PHP。不过从这取得的包可能会因为某个未经完整测试的代码提交而使PHP工作不正常。但是如果你想研究下PHP6.0的进展，这里绝对是你方便获取它的地方。 最后，你可以直接从版本库中导出此时此刻的源码。作为一个扩展开发者，从版本库或者snaps中获取php看起来并没有多大的作用，但是如果我们要将这个扩展推送到版本库中时，便需要熟练的掌握checkout和checkin的步骤了。签出的地址在上面已经说过了。

links

- 4 动手编译PHP
- 4.2 PHP编译前的config配置

4.2 动手编译PHP

第一章我们曾介绍过，PHP编译前的configure有两个特殊的选项，打开它们对我们开发PHP扩展或者进行PHP嵌入式开发时非常有帮助。但是当我们正常使用PHP的时候，则不应该开启这两个选项。

--enable-debug

顾名思义，它的作用是激活调试模式。它将激活PHP源码中几个非常关键的函数，最典型的功能便是在每一个请求结束后给出这一次请求中内存的泄漏情况。 回顾一下第三章《内存管理》部分，php内核中的ZendMM(Zend Memory Manager)将会在每一个请求结束后强制释放在这个请求中申请的内存。By running a series of aggressive regression tests against newly developed code, leak points can be easily spotted and plugged prior to any public release. Take a look at the following code snippet:

```
void show_value(int n)
{
    char *message = emalloc(1024);

    sprintf(message, "The value of n is %d\n", n);
    php_printf("%s", message);
}
```

上面的代码执行后，将会导致1024B的内存泄漏，但是在ZendMM的帮助下，在请求结束后会被PHP内核自动的释放掉。 但是如果你开启了--enable-debug选项，在请求结束后内核便会给出一条信息，告知我们程序猿这次请求的内存泄漏情况。 /cvs/php5/ext/sample/sample.c(33) :Freeing 0x084504B8 (1024 bytes), script=-

=== Total 1 memory leaks detected === 这条提示告知我们在这次请求结束后，ZendMM清理了泄漏的内存以及泄漏的内存位置。在它的帮助下，我们可以很快的定位到有问题的代码，然后通过efree等函数修正这个bug。 其实，内存泄漏并不是我们在开发中碰到的唯一错误，还有很多其它的bug很难被检测出来。有时候这些bug是致命的，但很难定位到出问题的代码。很多时候我们忙活了大半个晚上，修改了很多文件，最后make，但是当我们运行脚本的时候却得到下面的段错误。

```
$ sapi/cli/php -r 'myext_samplefunc();'
Segmentation Fault
//如果中文环境，则显示段错误
```

Orz...错误出在哪呢？我们遍历myext_samplefuc的所有实现代码也没有发现问题，扔进gdb里也仅仅显示几行无关紧要的信息而已。这种情况下，enable-debug就能帮你大忙了，打开这个选项后，你编译出来的php则会嵌入gdb或其它文件需要的所有调试信息。现在我们重新编译这个扩展，再扔进gdb里调试，便会得到如下的信息：

```
#0 0x1234567 php_myext_find_delimiter(str=0x1234567 "foo@#(FHVN)@\x98\xE0...",
                                      strlen=3, tsrm_ls=0x1234567)
p = strchr(str, ',');
```

现在所有的问题都水落石出了，字符串变量str没有以NULL结尾，而我们却把它当作一个参数传给了二进制不安全的字符串处理函数，str将会扫描str知道找到NULL为止，它的扫描肯定是越界了，然后引发了一个段错误。找到问题根源后，我们只要用memchr来替换strchr函数就能修复这个bug了。

--enable-maintainer-zts

第二个重要的参数便是激活php的线程安全机制(Thread Safe Resource Manager(TSRM)/Zend Thread Safety(ZTS))，使我们开发出的程序是线程安全的。对于TRSM的介绍大家可以参考第一章的介绍，在平时的开发中，建议打开这个选项。

--enable-embed

其实还有一个选项比较重要，那就是enable-embed，它主要用在你做php的嵌入式开发的场景中。平时我们把php作为apache的一个module进行编译，得到libphp5.so，而这个选项便使php编译后得到一个与我们设定的SAPI相对应的结果。

links

- 4.1 编译前的准备
- 4.3 Unix/Linux平台下的编译

4.3 动手编译PHP

编译之前如果需要了解一下php的configure脚本的各个配置，./configure --help一下即可，或者参考一下网络上的资料。当你确定了应该开启哪几个选项，选项都应该赋什么值后，便可以开始正式的编译我们的PHP了。这里假设你下载了php-5.3的源码，而且你将其解压到/php-5.3/目录下。进入终端，通过cd命令进入/php-5.3/目录，执行./configure脚本，然后make,make test,比如：

```
cd /php-5.3
./configure --prefix=/walu/php/ --enable-debug --enable-maintainer-zts
make
make test
make clean //自愿执行，非必须。
```

make，尤其是make test命令是个耗时大户，具体执行时间的长短与机器配置有关(这两个命令做练习可以，如果我们部署开发环境的时候，建议大家用apt-get或者yum来安装现成的)。

links

- 4.2 PHP编译前的config配置
- 4.4 在Win32平台上编译PHP

4.4 动手编译PHP

注意，没翻译的这节仅代表作者05年的观点。

```
As with the UNIX build, the first step to preparing a Windows build is to unpack the sour

Start by renaming the file back to php-5.1.0.tar.gz (if necessary). If you have a program

Whatever decompression program you use, have it decompress php-5.1.0.tar.gz to the root d

After it's unpacked, open up a build environment window by choosing Start, All Programs,

A simple command prompt window will open up stating the target build platform. This comma

Now, change the directory to the location where you unpacked PHP

C:\PHPDEV\php-5.1.0and run buildconf.bat.

C:\Program Files\Microsoft Platform SDK> cd \PHPDEV\php-5.1.0
C:\PHPDEV\php-5.1.0> buildconf.bat


If all is going well so far you'll see the following two lines of output:

Rebuilding configure.js
Now run 'cscript /nologo configure.js help'


At this point, you can do as the message says and see what options are available. The ena

In this example I've removed a few other extensions that aren't relevant to extension and

C:\php-5.1.0> cscript /nologo configure.js without-xml without-wddx \
without-simplexml without-dom without-libxml disable-zlib \
without-sqlite disable-odbc disable-cgi enable-cli \
enable-debug without-iconv


Again, a stream of informative output will scroll by, followed by instructions to execute

C:\php-5.1.0> nmake


Finally, a working build of PHP compiled for the Win32 platform.
```

links

- 4.3 Unix/Linux平台下的编译
- 4.5 第四章小结

4.5 动手编译PHP

单就开发一个最基本的php扩展来说，该掌握的前置知识我们已经都掌握了。在接下来的章节里我们将会深入的研究如何制作一个PHP扩展，以及制作一个优秀的PHP扩展所需的其它知识。 此外，如果你只想把PHP当作一个嵌入式应用来使用，我们也强烈的建议你不要直接跳到最后几章，因为在接下来的章节里我们将详细的介绍与PHP内核密切相关的一些内容，比如HashTable、数组、对象......等等的实现方式与应用方法。

links

- 4.4 在Win32平台上编译PHP
- 5 Your First Extension

5 Your First Extension

每一个PHP扩展都至少需要两个文件：一个配置文件和一个源文件。配置文件用来告诉编译器应该编译哪几个文件，以及编译本扩展是否需要的其它lib。

links

- 4.5 第四章小结
- 5.1 一个扩展的基本结构

5.1 Your First Extension

配置文件

才开始，我们先用最快的(不是最标准的)的方式来建立一个代码最少的扩展。在php源码文件夹的ext目录下创建一个新的文件，这里我取的名字叫做walu，它往往就是我们扩展的名字。其实这个文件夹可以放在任何一个位置，但是为了我们在后面介绍win32的编译与静态编译，我们还是把它放在php源码的ext目录下。现在，我们在这个目录下创建一个config.m4文件，并输入以下内容：

```
PHP_ARG_ENABLE(walu,
    [Whether to enable the "walu" extension],
    [  enable-walu        Enable "walu" extension support])

if test $PHP_WALU != "no"; then
    PHP_SUBST(WALU_SHARED_LIBADD)
    PHP_NEW_EXTENSION(walu, walu.c, $ext_shared)
fi
```

上面PHP_ARG_ENABLE函数有三个参数，第一个参数是我们的扩展名(注意不用加引号)，第二个参数是当我们运行./configure脚本时显示的内容，最后一个参数则是我们在调用./configure --help时显示的帮助信息。

> 也许有人会问，为什么有的扩展的开启方式是 --enable-extname的形式，有的则是--with-extname的形式呢？其实两者并没有什么本质的不同，只不过enable多代表不依赖外部库便可以直接编译，而with大多需要依赖于第三方的lib。现在，我们的扩展并不需要依赖其它的库文件，所以我们直接使用--enable-walu便可以了。在第17章的时候我们将接触通过CFLAGS和LDFLAGS来配置自己的扩展，使其依赖第三方库文件才能被编译成php扩展。

如果我们显示运行./configure --enable-walu，那么终端环境便会自动将$PHP_WALU变量设置为yes，而PHP_SUBST函数只不过是php官方对autoconf里的AC_SUBST函数的一层封装。最后重要的一点是，PHP_NEW_EXTENSION函数声明了这个扩展的名称、需要的源文件名、此扩展的编译形式。如果我们的扩展使用了多个文件，便可以将这多个文件名罗列在函数的参数里，如：

```
PHP_NEW_EXTENSION(sample, sample.c sample2.c sample3.c, $ext_shared)
```

最后的$ext_shared参数用来声明这个扩展不是一个静态模块，而是在php运行时动态加载的。

下面，我们来编写实现扩展主逻辑的源文件walu.c:

```
//加载config.h，如果配置了的话
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

//加载php头文件
#include "php.h"
```

```
#define phpext_walu_ptr &walu_module_entry

//module entry
zend_module_entry walu_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
     STANDARD_MODULE_HEADER,
#endif
    "walu", //这个地方是扩展名称，往往我们会在这个地方使用一个宏。
    NULL, /* Functions */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    "2.1", //这个地方是我们扩展的版本
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_WALU
ZEND_GET_MODULE(walu)
#endif
```

这就是所有的代码了，不过鉴于我们平时的开发习惯，往往会把这一份代码分成两份，一个.h文件，一个.c文件。上面的代码只是生成了一基本的框架，而没有任何实际的用处。 紧接着，创建一个zend_module_entry结构体，你肯定已经发现了，依据ZEND_MODULE_API_NO 是否大于等于20010901，这个结构体需要不同的定义格式。20010901大约代表PHP4.2.0版本，所以我们现在的扩展几乎都要包含STANDARD_MODULE_HEADER这个元素了。 其余六个成员我们可以先赋值为NULL，其实看看它们各自后面的注释你就应该大体上了解它们各自是负责哪一方面的工作了。 最后，最底下的代码用来标志我们的这个扩展是一个shared module。它是干么的呢？我也说不清楚，反正带上就对了，否则扩展会工作不正常。原文解释：This brief conditional simply adds a reference used by Zend when your extension is loaded dynamically. Don't worry about what it does or how it does it too much; just make sure that it's around or the next section won't work.

标准一些

根据我们平时的开发习惯，应该不会把所有代码都写在这一个文件里的，我们需要把上述代码放在两个文件里，一个头文件，一个c文件。

```
//php_walu.h
#ifndef WALU_H
#define WALU_H

//加载config.h，如果配置了的话
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

//加载php头文件
#include "php.h"
```

```
#define phpext_walu_ptr &walu_module_entry
extern zend_module_entry walu_module_entry;

#endif
```

下面的是c文件

```
//walu.c
#include "php_walu.h"
//module entry
zend_module_entry walu_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
     STANDARD_MODULE_HEADER,
#endif
    "walu", //这个地方是扩展名称，往往我们会在这个地方使用一个宏。
    NULL, /* Functions */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    "2.1", //这个地方是我们扩展的版本
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_WALU
ZEND_GET_MODULE(walu)
#endif
```

links

- 5 Your First Extension
- 5.2 编译我们的扩展

5.2 Your First Extension

我们已经在上一节准备好了需要编译的源文件，接下来需要的便是把它们编译成目标文件了。因为在*nix平台和win平台下的编译步骤有些差异，所以这个地方需要分成两块介绍，很不幸，win部分还没有整理，请随时关注本项目。

在*nix下编译

第一步：我们需要根据config.m4文件生成一个configure脚本、Makefile等文件，这一步有phpize来帮我们做：

```
$ phpize
PHP Api Version: 20041225
Zend Module Api No: 20050617
Zend Extension Api No: 220050617
```

The extra 2 at the start of Zend Extension Api No isn't a typo; it corresponds to the Zend Engine 2 version and is meant to keep this API number greater than its ZE1 counterpart.
现在查看一下我们扩展所在的目录，会发现多了许多文件。phpize程序根据config.m4里的信息生成了许多编译php扩展必须的文件，比如生成makefiles等，这为我们省了很多的麻烦。 接下来我们运行./configure脚本，这里我们并不需要再注明enable-maintainer-zts、enable-debug等参数，phpize程序会自动的去已经编译完成的php核心里获取这几个参数的值。 接下来就像我们安装其它程序一样执行make; make test;即可，如果没有错误，那么在module文件夹下面便会生成我们的目标文件 —— walu.so。

在windows平台下编译

The config.m4 file you created earlier was actually specific to the *nix build. In order to make your extension compile under Windows, you'll need to create a separatebut similarconfiguration file for it. Add config.w32 with the following contents to your ext/sample directory:

```
ARG_ENABLE("sample", "enable sample extension", "no");
if (PHP_SAMPLE != "no") {
    EXTENSION("sample", "sample.c");
}
```

As you can see, this file bears a resemblance on a high level to config.m4. The option is declared, tested, and conditionally used to enable the build of your extension. Now you'll repeat a few of the steps you performed in Chapter 4, "Setting Up a Build Environment," when you built the PHP core. Start by opening up a build window from the Start menu by selecting All Programs, Microsoft Platform SDK for Windows Server 2003 SP1, Open Build Environment Window, Windows 2000 Build Environment, Set Windows 2000 Build Environment (Debug), and running the C:\Program Files\Microsoft Visual Studio 8\VC\bin\vcvars32.bat batch file. Remember, your installation might require you to select a different build target or run a slightly different batch file. Refer to the notes in the corresponding section of Chapter 4 to refresh your memory. Again, you'll want to go to the root of your build directory and rebuild the configure

script.

```
C:\Program Files\Microsoft Platform SDK> cd \PHPDEV\php-5.1.0
C:\PHPDEV\php-5.1.0> buildconf.bat
Rebuilding configure.js
Now run 'cscript /nologo configure.js help'
```

This time, you'll run the configure script with an abridged set of options. Because you'll be focusing on just your extension and not the whole of PHP, you can leave out options pertaining to other extensions; however, unlike the Unix build, you do need to include the enable-debug switch explicitly even though the core build already has it. The only crucial switch you'll need hereapart from debug of courseis enable-sample=shared. The shared option is required here because configure.js doesn't know that you're planning to build sample as a loadable extension. Your configure line should therefore look something like this:

```
C:\PHPDEV\php-5.1.0> cscript /nologo configure.js \
enable-debug enable-sample=shared
```

Recall that enable-maintainer-zts is not required here as all Win32 builds assume that ZTS must be enabled. Options relating to SAPIssuch as embedare also not required here as the SAPI layer is independent from the extension layer.
Lastly, you're ready to build the extension. Because this build is based from the coreunlike the Unix extension build, which was based from the extensionyou'll need to specify the target name in your build line.

C:\PHPDEV\php-5.1.0> nmake php_sample.dll Once compilation is complete, you should have a working php_sample.dll binary ready to be used in the next step. Remember, because this book focuses on *nix development, the extension will be referred to as sample.so rather than php_sample.dll in all following text. Loading an Extension Built as a Shared Module

加载扩展

为了使PHP能够找到需要的扩展文件，我们需要把编译好的so文件或者dll文件复制到PHP的扩展目录下，它的地址我们可以通过phpinfo()输出的信息找到，也可以在php.ini文件里进行配置找到并配置，名称为：extension_dir的值。默认情况下，php.ini文件位于/usr/local/lib/php.ini或者C:\windows\php.ini(现在由于fastcgi模式居多，在win平台上php.ini越来越多的直接存在于php-cgi.exe程序所在目录下)。如果找不到，我们可以通过php -i 命令或者<?php phpinfo();来查看当前加载的php.ini文件位置。 一旦我们设置了extension_dir，便可以在我们的web文件中引用我们的扩展了，我们可以通过dl命令来将我们的扩展加载到内存中来。

```php
<?php
    dl('sample.so');
    var_dump(get_loaded_extensions());
?>
```

如果在输出中我们没有找到walu.so，那肯定是哪里出问题了。这时候我们需要根据程序的输出信息去查找

错误。 上面这样每次使用扩展都需要先dl一下真是太麻烦了，其实我们有更好的办法让php运行时自动加载我们的扩展。那就是在php.ini里这样配置：

```
extension_dir=/usr/local/lib/php/modules/
extension=walu.so
```

这样只要我们把walu.so这个文件放置在extension_dir配置的目录下，php就会在每次启动的时候自动加载了。这样我们就可以像我们平时使用curl、Mysql扩展一样直接使用，而不用麻烦的调用dl函数了。 备注：以下的章节我们都默认使用上面的这种方式来加载我们的扩展，而不是调用dl函数。

links

- 5.1 一个扩展的基本结构
- 5.3 静态编译

5.3 Your First Extension

我们检查一下PHP语言中get_loaded_extensions()函数的输出，会发现有一些扩展并没有php.ini文件中调用，而它们确实也已经加载到PHP里去了，可以让我们在PHP语言中使用，如standard、Reflection、Core等。它们便是静态编译的，它们没有被编译成so或者dll文件供PHP动态调用，而是直接和PHP主程序编译到一起。

在*nix上执行静态编译

现在，先让我们执行一下PHP源码根目录下的./configure --help命令。会发现输出信息并没有包含我们的扩展，这是因为这个configure脚本生成的时候，我们的扩展还没有编写呢。(这个configure是PHP官方分发的。)，所以首先我们需要使用buildconf命令生成新的configure脚本。 $ ./buildconf --force

If you're using a production release of PHP to do development against, you'll find that ./buildconf by itself doesn't actually work. In this case you'll need to issue: ./buildconf force to bypass some minor protection built into the ./configure command.
现在当我们再执行./configure --help的时候，便会发现walu扩展的信息已经出现了。现在我们只需要重新走一遍PHP的编译过程，便可以把我们的扩展以静态编译的方式加入到PHP主程序中了。哦，千万不要忘记使用--enable-walu参数开启我们的扩展。 当然，对于我们学习如何开发PHP扩展来讲，静态编译可不是一个好主意，因为如果采用静态编译的方式，只要我们的扩展做了改动，便需要重新编译整个PHP才行，这个过程太痛苦了。还是用前一节的方式吧。但是这种方式有利于提高性能，所以如果我们是在部署生产环境，则可以考虑！

Building Statically Under Windows

Regenerating the configure.js script for Windows follows the same pattern as regenerating the ./configure script for *nix. Navigate to the root of the PHP source tree and reissue buildconf.bat as you did in Chapter 4. The PHP build system will scan for config.w32 files, including the one you just made for ext/sample, and generate a new configure.js script with which to build a static php binary.

links

- 5.2 编译我们的扩展
- 5.4 编写函数

5.4 Your First Extension

前面我们已经生成好了一份扩展框架，但它是没有什么实际作用的。一个扩展的作用可大了去了，既可以操作PHP中的变量、常量，还可以定义函数、类、方法、资源等。先让我们从函数说起吧！

ZEND_FUNCTION()宏函数

ZEND_FUNCTION()宏函数也可以写成PHP_FUNCTION()，但ZEND_FUNCTION()更前卫、标准一些，但两者是完全相同的。

```
#define PHP_FUNCTION                ZEND_FUNCTION

#define ZEND_FUNCTION(name)             ZEND_NAMED_FUNCTION(ZEND_FN(name))
#define ZEND_NAMED_FUNCTION(name)     void name(INTERNAL_FUNCTION_PARAMETERS)
#define ZEND_FN(name)                  zif_##name
```

其中，zif是zend internal function的意思，zif_前缀是可供PHP语言调用的函数在C语言中的函数名称前缀。

```
ZEND_FUNCTION(walu_hello)
{
    php_printf("Hello World!\n");
}
```

上面定义了一个函数，在C语言中展开后应该是这样的:

```
void zif_walu_hello(INTERNAL_FUNCTION_PARAMETERS)
{
    php_printf("Hello World!\n");
}
```

上面的展开式仅供参考，绝不推荐在编程时使用，我们应该采用宏的形式，来提高程序的兼容性与可读性。 上面的代码定义了一个可供用户在PHP语言中调用的函数实现，但现在用户还不能在程序中调用，因为这个函数还没有与用户端建立联系，也就是说虽然我们在C中完成了它的实现，但用户端PHP语言还根本不知道它的存在呢。 现在我们回头看一下5.1节中我们为扩展定义的zend_module_entry walu_module_entry（它是联系C扩展与PHP语言的重要纽带）中的“NULL, / *Functions* /”，当时我们为它赋予了NULL，是因为还没有函数，现在我们已经为它编写了函数了，便可以给它赋予一个新值了，这个值需要是zend_function_entry[]类型的，首先让我们来构造这个重要数据。

```
static zend_function_entry walu_functions[] = {
    ZEND_FE(walu_hello,        NULL)
    { NULL, NULL, NULL }
};

/*
下面是ZEND_FE的定义
```

```
#define ZEND_FE(name, arg_info)        ZEND_FENTRY(name, ZEND_FN(name), arg_info, 0)
#define ZEND_FENTRY(zend_name, name, arg_info, flags)    { #zend_name, name, arg_info, (z

ZEND_FE(walu_hello,        NULL)展开后便是：
{"walu_hello",zif_walu_hello,NULL, (zend_uint) (sizeof(NULL)/sizeof(struct _zend_arg_info

*/
```

其中最后的{NULL,NULL,NULL}是固定不变的。ZEND_FE()宏函数是对我们walu_hello函数的一个声明，如果我们有多个函数，可以直接以类似的形式添加到{NULL,NULL,NULL}之前，注意每个之间不需要加逗号。 其中的arg_info我们现在先赋予NULL就行了，我们将在第7章讨论这个参数。确保一切无误后，我们替换掉zend_module_entry里的原有成员，现在应该是这样的：

```
ZEND_FUNCTION(walu_hello)
{
    php_printf("Hello World!\n");
}

static zend_function_entry walu_functions[] = {
    ZEND_FE(walu_hello,        NULL)
    { NULL, NULL, NULL }
};

zend_module_entry walu_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
     STANDARD_MODULE_HEADER,
#endif
    "walu", //这个地方是扩展名称，往往我们会在这个地方使用一个宏。
    walu_functions, /* Functions */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    "2.1", //这个地方是我们扩展的版本
#endif
    STANDARD_MODULE_PROPERTIES
};
```

现在configure、make、make test，复制到extension dir。用下面这个命令来测试下，应该会输出hello world了，如果没有输出，说明你哪个地方做错了，查不出来的话可以给我发mail，看看是不是特例:-)

```
$ php -r 'walu_hello();'
```

Zend Internal Functions

zif前缀在前面我们已经说过了，代表着*"Zend Internal Function"*，主要用来避免命名冲突，比如*PHP*语言中有个*strlen()*函数，而*C*语言中也有*strlen()*函数，所以*PHP*中的*strlen*在*C*中的实现不能是*strlen*，而应改是一个不同的名字。 但是有些时候尽管我们加了*zif*前缀，还会出现一些冲突问题。比如函数名称本身是一个宏

名称从而被编译器替换掉了。在这种情况下，我们需要手动来为我们扩展中的函数命名，这一步操作通过ZEND_NAMED_FUNCTION(diy_walu_hello)来代替ZEND_FUNCTION(hello_hello)。前者由我们指定名称，后者自己加上前缀。 如果我们在定义函数时使用了ZEND_NAMED_FUNCTION()，那么在walu_functions[]里，我们需要用ZEND_NAMED_FE()宏来代替ZEND_FE()宏。即：ZEND_NAMED_FE(walu_hello,diy_walu_hello,NULL) 上面的技术在ext/standard/file.c用到了，我们可以看fopen()函数的定义：PHP_NAMED_FUNCTION(php_if_fopen)。但是用户端不会感觉到任何变化，还是用fopen函数来使用，因为zend_function_entry中每一项的第一个值代表这此函数在PHP语言中的名称。 Internally, however, the function is protected from being mangled by preprocessor macros and over-helpful compilers.（原作者说的这个理由我也没看明白，请知者指点）

Function Aliases

去PHP手册里查一下pos()函数，会得到这么一条信息："This function is an alias of: current()";也就是说，它只是current的一个软链接而已，类似linux中的ln -s命令，理解成win下的快捷方式也成。运行pos函数，其实就是在运行current函数，转接了一下而已。这往往是因为版本升级引起的，新版本中的程序提供了某个功能的新的实现，先为原来的函数改个名，但还需要保留原来的函数名，所以这就用到了alias。这个功能可以在内核中通过ZEND_NAMED_FE宏来实现。

```
static zend_function_entry walu_functions[] = {
    ZEND_FE(walu_hello,        NULL)
    ZEND_NAMED_FE(walu_hi,    ZEND_FN(walu_hello),    NULL)
    { NULL, NULL, NULL }
};

/*
ZEND_NAMED_FE也可以写成PHP_NAMED_FE,但推荐用前者
#define ZEND_NAMED_FE(zend_name, name, arg_info)    ZEND_FENTRY(zend_name, name, arg_info
*/
```

通过ZEND_NAMED_FE的展开式我们了解到，它只是把PHP语言中的两个函数的名字对应到同一个C语言函数而已。 其实还有另外一种写法：

```
static zend_function_entry walu_functions[] = {
    ZEND_FE(walu_hello,        NULL)
    ZEND_FALIAS(walu_hi,walu_hello,    NULL)
    { NULL, NULL, NULL }
};

/*
#define ZEND_FALIAS(name, alias, arg_info)            ZEND_FENTRY(name, ZEND_FN(alias), a
*/
```

展开式是一样的，真不清楚官方鼓捣这么多同样的宏干啥。

```
<?php
walu_hi();
walu_hello();
```

links

- 5.3 静态编译
- 5.5 第五章小结

5.5 Your First Extension

在这一章里，我们学会了如何创建一个PHP框架并为其添加函数，并编译到PHP中供用户在PHP语言中调用。在接下来的章节里，我们将陆续看到许多高级的PHP内核特性，从而使我们编写出更好的PHP扩展。编译PHP源码的环境会随着平台与时间的不同而变化，如果本章讲述的知识无法使你顺利的编译PHP，那你可以给我发信，或者去php.net寻找答案，当然最简单的方法是Google，切记的是，万一Google抽风，不要忘了还有Baidu。

links

- 5.4 编写函数
- 6 函数返回值

6 函数返回值

本章目录

PHP语言中函数的返回值是通过return来完成的，就像下面的程序：

```
<?php
function sample_long() {
  return 42;
}
$bar = sample_long();
```

C语言也一样使用return关键字

```
int sample_long(void) {
    return 42;
}
int main(void) {
    int bar = sample_long();
    return 1;
}
```

那我们在扩展中编写的PHP函数如何把返回值回馈给用户端的函数调用者呢？看好，这里指的是回馈给，而不是单单的return～

links

- 5.5 第五章小结
- 6.1 一个特殊的参数：return_value

6.1 函数返回值

你也许会认为扩展中定义的函数应该直接通过return关键字来返回一个值，比如由你自己来生成一个zval并返回，就像下面这样：

```
ZEND_FUNCTION(sample_long_wrong)
{
    zval *retval;

    MAKE_STD_ZVAL(retval);
    ZVAL_LONG(retval, 42);

    return retval;
}
```

但是，上面的写法是无效的！与其让扩展开发员每次都初始化一个zval并return之，zend引擎早就准备好了一个更好的方法。它在每个zif函数声明里加了一个zval*类型的形参，名为return_value，专门来解决返回值这个问题。在前面我们已经知道了ZEND_FUNCTION宏展开后是void name(INTERNAL_FUNCTION_PARAMETERS)的形式，现在是我们展开代表参数声明的INTERNAL_FUNCTION_PARAMETERS宏的时候了。

```
#define INTERNAL_FUNCTION_PARAMETERS int ht, zval *return_value, zval **return_value_ptr,
```

- int ht
- zval *return_value，我们在函数内部修改这个指针，函数执行完成后，内核将把这个指针指向的zval返回给用户端的函数调用者。
- zval **return_value_ptr，
- zval *this_ptr，如果此函数是一个类的方法，那么这个指针的含义和PHP语言中$this变量差不多。
- int return_value_used，代表用户端在调用此函数时有没有使用到它的返回值。

下面让我们先试验一个非常简单的例子，我先给出PHP语言中的实现，然后给出我们在扩展中用C语言完成相同功能的代码。 ````php ```` 下面是我们在编写扩展时的实现。 ````c ZEND_FUNCTION(sample_long) { ZVAL_LONG(return_value, 42); return; } ```` 需要注意的是，ZEND_FUNCTION本身并没有通过return关键字返回任何有价值的东西，它只不过是在运行时修改了return_value指针所指向的变量的值而已，而内核则会把return_value指向的变量作为用户端调用此函数后的得到的返回值。回想一下,ZVAL_LONG()宏是对一类操作的封装，展开后应该就是下面这样： ````c Z_TYPE_P(return_value) = IS_LONG; Z_LVAL_P(return_value) = 42; //更彻底的讲，应该是这样的： return_value->type = IS_LONG; return_value->value.lval = 42; ````
我们千万不要自己去修改return_value的is_ref__gc和refcount__gc属性，这两个属性的值会由PHP内核自动管理。
现在我们把它加到我们在第五章得到的那个扩展框架里，并把这个函数名称注册到函数入口数组里，就像下面这样： ````c static zend_function_entry walu_functions[] = { ZEND_FE(walu_hello, NULL) PHP_FE(sample_long, NULL) { NULL, NULL, NULL } }; ```` 现在我们编译我们的扩展，便可以在用户端通过调用sample_long函数来得到一个整型的返回值了: ````php ```` ### 与return_value有关的宏 return_value

如此重要，内核肯定早已经为它准备了大量的宏，来简化我们的操作，提高程序的质量。 在前几章我们接触的宏大多都是以ZVAL_开头的，而接下来我们要介绍的宏的名字是：RETVAL。 再回到上面的那个例子，我们用RETVAL来重写一下： ```c PHP_FUNCTION(sample_long) { RETVAL_LONG(42); //展开后相当与ZVAL_LONG(return_value, 42); return; } ```` 大多数情况下，我们在处理完return_value后所做的便是用return语句结束我们的函数执行，帮人帮到底，送佛送到西，为了减少我们的工作量，内核中还提供了RETURN_*系列宏来为我们自动补上return;如： PHP_FUNCTION(sample_long) { RETURN_LONG(42); //#define RETURN_LONG(l) { RETVAL_LONG(l); return; } php_printf("I will never be reached.\n"); //这一行代码永远不会被执行。 } 下面，我们给出目前所有的RETVAL_***宏和RETURN_***宏，供大家查阅使用。 ```c //这些宏都定义在Zend/zend_API.h文件里 #define RETVAL_RESOURCE(l) ZVAL_RESOURCE(return_value, l) #define RETVAL_BOOL(b) ZVAL_BOOL(return_value, b) #define RETVAL_NULL() ZVAL_NULL(return_value) #define RETVAL_LONG(l) ZVAL_LONG(return_value, l) #define RETVAL_DOUBLE(d) ZVAL_DOUBLE(return_value, d) #define RETVAL_STRING(s, duplicate) ZVAL_STRING(return_value, s, duplicate) #define RETVAL_STRINGL(s, l, duplicate) ZVAL_STRINGL(return_value, s, l, duplicate) #define RETVAL_EMPTY_STRING() ZVAL_EMPTY_STRING(return_value) #define RETVAL_ZVAL(zv, copy, dtor) ZVAL_ZVAL(return_value, zv, copy, dtor) #define RETVAL_FALSE ZVAL_BOOL(return_value, 0) #define RETVAL_TRUE ZVAL_BOOL(return_value, 1) #define RETURN_RESOURCE(l) { RETVAL_RESOURCE(l); return; } #define RETURN_BOOL(b) { RETVAL_BOOL(b); return; } #define RETURN_NULL() { RETVAL_NULL(); return;} #define RETURN_LONG(l) { RETVAL_LONG(l); return; } #define RETURN_DOUBLE(d) { RETVAL_DOUBLE(d); return; } #define RETURN_STRING(s, duplicate) { RETVAL_STRING(s, duplicate); return; } #define RETURN_STRINGL(s, l, duplicate) { RETVAL_STRINGL(s, l, duplicate); return; } #define RETURN_EMPTY_STRING() { RETVAL_EMPTY_STRING(); return; } #define RETURN_ZVAL(zv, copy, dtor) { RETVAL_ZVAL(zv, copy, dtor); return; } #define RETURN_FALSE { RETVAL_FALSE; return; } #define RETURN_TRUE { RETVAL_TRUE; return; } ```` 其实，除了这些标量类型，还有很多php语言中的复合类型我们需要在函数中返回，如数组和对象，我们可以通过RETVAL_ZVAL与RETURN_ZVAL来操作它们，有关它们的详细介绍我们将在后续章节中叙述。 ### 不返回值可以么？ 其实，zend internal function的形参中还有一个比较常用的名为return_value_used的参数，它是干嘛使的呢？它用来标志这个函数的返回值在用户端有没有用到。看下面的代码： ```php 在上面的代码中，$b其实是$a的一个引用，当最后一行代码执行后，$a和$b都开始寻找'bar'这个字符串对应的zval，让我们以内核的角度重新观察这一切： #if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && PHP_MINOR_VERSION > 0) ZEND_FUNCTION(return_by_ref) { zval **a_ptr; zval *a; //检查全局作用域中是否有$a这个变量，如果没有则添加一个 //在内核中真的是可以胡作非为啊，:-) if(zend_hash_find(&EG(symbol_table) , "a",sizeof("a"),(void **)&a_ptr) == SUCCESS) { a = *a_ptr; } else { ALLOC_INIT_ZVAL(a); zend_hash_add(&EG(symbol_table), "a", sizeof("a"), &a,sizeof(zval*), NULL); } //废弃return_value,使用return_value_ptr来接替它的工作 zval_ptr_dtor(return_value_ptr); if(!a->is_ref__gc && a->refcount__gc > 1) { zval *tmp; MAKE_STD_ZVAL(tmp); *tmp = *a; zval_copy_ctor(tmp); tmp->is_ref__gc = 0; tmp->refcount__gc = 1; zend_hash_update(&EG(symbol_table), "a", sizeof("a"), &tmp,sizeof(zval*), NULL); a = tmp; } a->is_ref__gc = 1; a->refcount__gc++; *return_value_ptr = a; } #endif /* PHP >= 5.1.0 */ ```` return_value_ptr是定义zend internal function时的另外一个重要参数，他是一个zval**类型的指针，并且指向函数的返回值。我们调用zval_ptr_dtor()函数后，默认的return_value便被废弃了。这里的$a变量如果是与某个非引用形式的变量共用一个zval的话，便要进行分离。 不幸的是，如果你编译上面的代码，使用的时候便会得到一个段错误。为了使它能够正常的工作，需要在源文件中加一些东西： ```c #if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && PHP_MINOR_VERSION > 0) ZEND_BEGIN_ARG_INFO_EX(return_by_ref_arginfo, 0, 1, 0) ZEND_END_ARG_INFO () #endif /* PHP >= 5.1.0 */ 然后使用下面的代码来申明我们的定义的函数： #if (PHP_MAJOR_VERSION > 5) || (PHP_MAJOR_VERSION == 5 && PHP_MINOR_VERSION > 0) ZEND_FE(return_by_ref,

return_by_ref_arginfo) #endif /* PHP >= 5.1.0 */ ```` arginfo是一种特殊的结构体，用来提前向内核告知此函数具有的一些特定的性质，如本例，其将告诉内核本函数需要引用形式的返回值，所以内核不再通过return_value来获取执行结果，而是通过return_value_ptr。如果没有arginfo，那内核会预先把return_value_ptr置为NULL，当我们对其调用zval_ptr_dtor()函数时便会使程序崩溃。
这一些代码都包含在了一个宏里面，只有在php版本大于等于5.1的时候才会被启用。如果没有这些if、endif，那我们的程序将无法在php4下通过编译，在php5.0上也会激活一些无法预测的错误。

- zone(zqx10104#163.com)于2011-10-20提供了一个Bug，:-)

links

- 6 函数返回值
- 6.2 引用与函数的执行结果

6.2 函数返回值

一个函数的执行结果要返回给调用者，除了使用return功能，还有一种办法，那就是以引用的形式传递参数，然后在内部修改这个参数的值。前一种方法往往只能返回一个值，如果我们的函数执行结果具有多种数据，便需要把这些数据打包到一个数组、类等复合类型的变量中才能得以实现；但后一种方法相比而言就简单一些了。

运行时传递引用：Call-time Pass-by-ref

标题有点绕口，其实很简单，功能如以下php代码所示：

```
<?php
function byref_calltime($a) {
    $a = '(modified by ref!)';
}

$foo = 'I am a string';

//使用&传递引用
byref_calltime(&$foo);
echo $foo;
//输出'(modified by ref!)'
```

我们在传递参数的时候使用&操作符，便可以传递$foo变量的引用过去，而不是copy一份。当我们在函数内核修改这个参数时，函数外部的$foo也跟着被一起修改了。同样的功能我们如何在扩展里实现呢，其实很简单，请看下面的源码：

```
ZEND_FUNCTION(byref_calltime)
{
    zval *a;

    //我们我接收的参数传给zval *a;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z", &a) == FAILURE)
    {
        RETURN_NULL();
    }

    //如果a不是以引用的方式传递的。
    if (!a->is_ref__gc)
    {
            return;
    }

    //将a转成字符串
    convert_to_string(a);

    //更改数据
    ZVAL_STRING(a," (modified by ref!)",1);
    return;
}
```

编译时的传递引用Compile-time Pass-by-ref

如果每一次都在调用函数时候都对参数加一个&符号真是太罗嗦了，有没有一个简单的办法呢，比如在定义函数的时候便声明这个参数是引用形式的，而不用用户自己加&符号表示引用，而由内核来完成这步操作？这个功能是有的，我们在PHP语言中可以这样实现。

```
<?php
在定义函数参数的时候加了引用符
function byref_compiletime(&$a) {
    $a = ' (modified by ref!)';
}
$foo = 'I am a string';

//这个地方我们没有加&引用符
byref_compiletime($foo);
echo $foo;
//输出 (modified by ref!)
```

上面的代码中，我们只是把引用符号从函数调用里转移到函数定义里。此功能在扩展里面实现的话就颇费周折了，我们需要提前为它定义一个arginfo结构体来向内核通知此函数的这个特定行为。添加此函数到module_entry里需要这样：

```
ZEND_FE(byref_compiletime, byref_compiletime_arginfo)
```

byref_compiletime_arginfo是一个arginfo结构体，我们在前面的章节中已经用过一次了。

原书中此处有arginfo在PHP4里的实现，被我略去了。
在Zend Engine 2 (PHP5+)中，arginfo的数据是由多个zend_arg_info结构体构成的数组，数组的每一个成员即每一个zend_arg_info结构体处理函数的一个参数。zend_arg_info结构体的定义如下：

```
typedef struct _zend_arg_info {
    const char *name;                /* 参数的名称*/
    zend_uint name_len;                /* 参数名称的长度*/
    const char *class_name;            /* 类名 */
    zend_uint class_name_len;        /* 类名长度*/
    zend_bool array_type_hint;        /* 数组类型提示 */
    zend_bool allow_null;            /* 是否允许为NULL  */
    zend_bool pass_by_reference;    /* 是否引用传递 */
    zend_bool return_reference;        /* 返回值是否为引用形式 */
    int required_num_args;            /* 必要参数的数量 */
} zend_arg_info;
```

生成zend_arg_info结构的数组比较繁琐，为了方便PHP扩展开发者，内核已经准备好了相应的宏来专门处理此问题，首先先用一个宏函数来生成头部，然后用第二个宏生成具体的数据，最后用一个宏生成尾部代码。

```
#define ZEND_BEGIN_ARG_INFO(name, pass_rest_by_reference)    ZEND_BEGIN_ARG_INFO_EX(name,
#define ZEND_BEGIN_ARG_INFO_EX(name, pass_rest_by_reference, return_reference, required_n
    static const zend_arg_info name[] = {
```

```
        { NULL, 0, NULL, 0, 0, 0, pass_rest_by_reference, return_reference, required_num_


#define ZEND_ARG_INFO(pass_by_ref, name)        { #name, sizeof(#name)-1, NULL, 0, 0, 0,
#define ZEND_ARG_PASS_INFO(pass_by_ref)            { NULL, 0, NULL, 0, 0, 0, pass_by_ref,
#define ZEND_ARG_OBJ_INFO(pass_by_ref, name, classname, allow_null) { #name, sizeof(#name
#define ZEND_ARG_ARRAY_INFO(pass_by_ref, name, allow_null) { #name, sizeof(#name)-1, NULL


#define ZEND_END_ARG_INFO()        };

//这里我们先看
ZEND_BEGIN_ARG_INFO(name, pass_rest_by_reference)
ZEND_BEGIN_ARG_INFO_EX(name, pass_rest_by_reference, return_reference,required_num_args)
```

这两个宏函数的前两个参数的含义是一样的，name便是这个zend_arg_info数组变量的名字，这里我们定义它为：byref_compiletime_arginfo。pass_rest_by_reference如果被赋值为1，则代表着所有的参数默认都是需要以引用的方式传递的(在arginfo中单独声明的除外)。而对于ZEND_BEGIN_ARG_INFO_EX的后两个参数：

- name和pass_rest_by_reference的含义同上。
- return_reference：声明这个函数的返回值需要以引用的形式返回，这个参数已经在前面章节用过了。
- required_num_args：函数被调用时，传递参数至少为前N个函数(也就是后面参数都有默认值)，当设置为-1时，必须传递所有参数

接下来让我们看生成具体数据的宏：

```
ZEND_ARG_PASS_INFO(by_ref)
//强制所有参数使用引用的方式传递


ZEND_ARG_INFO(by_ref, name)
//如果by_ref为1，则名称为name的参数必须以引用的方式传递，


ZEND_ARG_ARRAY_INFO(by_ref, name, allow_null)
ZEND_ARG_OBJ_INFO(by_ref, name, classname, allow_null)
这两个宏实现了类型绑定，也就是说我们在传递某个参数时，必须是数组类型或者某个类的实例。如果最后的参数为真，

//我们组合起来使用：
ZEND_BEGIN_ARG_INFO(byref_compiletime_arginfo, 0)
    ZEND_ARG_PASS_INFO(1)
ZEND_END_ARG_INFO()
```

为了使我们的扩展能够兼容PHP4，还需要使用#ifdef进行特殊处理。

```
#ifdef ZEND_ENGINE_2
    ZEND_BEGIN_ARG_INFO(byref_compiletime_arginfo, 0)
        ZEND_ARG_PASS_INFO(1)
    ZEND_END_ARG_INFO()
#else /* ZE 1 */
```

```
static unsigned char byref_compiletime_arginfo[] =   { 1, BYREF_FORCE };
#endif
```

我们copy一份ZEND_FUNCTION(byref_calltime)的实现，并重名成ZEND_FUNCTION(byref_compiletime)就行了。或者直接弄个ZEND_FALIAS就行了：

```
ZEND_FALIAS(byref_compiletime,byref_calltime,byref_compiletime_arginfo)
```

links

- 6.1 一个特殊的参数：return_value
- 6.3 第六章小结

6.3 函数返回值

在这一章里，我们集中讨论了如何把函数执行的结果返回给调用者，通过return语句、引用返回、通过参数返回等等，而且还初步了解了一下zend_arg_info。在下面的章节中，我们将去看一下内核是如何接收调用者传递的参数的。

links

- 6.2 引用与函数的执行结果
- 7 函数的参数

7 函数的参数

前面的章节我们look了一下如何在扩展中定义函数，它们的实现大都比较简单，但是在实际工作中，肯定会碰到函数接收参数的问题，而它就是我们这一章要讲解的内容。

links

- 6.3 第六章小结
- 7.1 zend_parse_parameters

7.1 函数的参数

最简单的获取函数调用者传递过来的参数便是使用zend_parse_parameters()函数。zend_parse_parameters()函数的前几个参数我们直接用内核里宏来生成便可以了，形式为：ZEND_NUM_ARGS() TSRMLS_CC，注意两者之间有个空格，但是没有逗号。从名字可以看出，ZEND_NUM_ARGS()代表着参数的个数。紧接着需要传递给zend_parse_parameters()函数的参数是一个用于格式化的字符串，就像printf的第一个参数一样。下面表示了最常用的几个符号。

```
type_spec是格式化字符串，其常见的含义如下：
参数    代表着的类型
b     Boolean
l     Integer 整型
d     Floating point 浮点型
s     String 字符串
r     Resource 资源
a     Array 数组
o     Object instance 对象
O     Object instance of a specified type 特定类型的对象
z     Non-specific zval 任意类型～
Z     zval**类型
f     表示函数、方法名称，PHP5.1里貌似木有... ...
```

这个函数就像printf()函数一样，后面的参数是与格式化字符串里的格式一一对应的。一些基础类型的数据会直接映射成C语言里的类型。

```
ZEND_FUNCTION(sample_getlong) {

    long foo;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,"l", &foo) == FAILURE)
    {
        RETURN_NULL();
    }
    php_printf("The integer value of the parameter is: %ld\n", foo);
    RETURN_TRUE;
}
```

```
一般来说，int和long这两种数据类型的数据往往是相同的，但也有例外情况。所以我们不应改把long的数组放在一个i
```



```
参数     对应C里的数据类型
b     zend_bool
l     long
d     double
s     char*, int 前者接收指针，后者接收长度
r     zval*
a     zval*
o     zval*
O     zval*, zend_class_entry*
```

```
z    zval*
Z    zval**
```

注意，所有的PHP语言中的复合类型参数都需要zval*类型来作为载体，因为它们都是内核自定义的一些数据结构。我们一定要确认参数和载体的类型一致，如果需要，它可以进行类型转换，比如把array转换成stdClass对象。 s和O(字母大写欧)类型需要单独说一些，因为它们都需要两个载体。我们将在接下来的章节里了解php中对象的具体实现。这样我们改写一下我们在第五章定义的一个函数：

```php
<?php
function sample_hello_world($name) {
    echo "Hello $name!\n";
}
```

在编写扩展时，我们需要用zend_parse_parameters()来接收这个字符串:

```c
ZEND_FUNCTION(sample_hello_world) {
    char *name;
    int name_len;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",&name, &name_len) == FAILURE
    {
        RETURN_NULL();
    }
    php_printf("Hello ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

如果传递给函数的参数数量小于zend_parse_parameters()要接收的参数数量，它便会执行失败，并返回FAILURE。
如果我们需要接收多个参数，可以直接在zend_parse_paramenters()的参数里罗列接收载体便可以了，如：

```php
<?php
function sample_hello_world($name, $greeting) {
    echo "Hello $greeting $name!\n";
}
sample_hello_world('John Smith', 'Mr.');
```

在PHP扩展里应该这样来实现：

```c
ZEND_FUNCTION(sample_hello_world) {
    char *name;
    int name_len;
    char *greeting;
    int greeting_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",&name, &name_len, &greeting
        RETURN_NULL();
```

```
    }
    php_printf("Hello ");
    PHPWRITE(greeting, greeting_len);
    php_printf(" ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

除了上面定义的参数，还有其它的三个参数来增强我们接收参数的能力,如下：

```
Type Modifier    Meaning
|         它之前的参数都是必须的，之后的都是非必须的，也就是有默认值的。
!         如果接收了一个PHP语言里的null变量，则直接把其转成C语言里的NULL，而不是封装成IS_NULL类型的zva
/         如果传递过来的变量与别的变量共用一个zval，而且不是引用，则进行强制分离，新的zval的is_ref__gc=
```

函数参数的默认值

现在让我们继续改写sample_hello_world(), 接下来我们使用一些参数的默认值，在php语言里就像下面这样：

```php
<?php
function sample_hello_world($name, $greeting='Mr./Ms.') {
    echo "Hello $greeting $name!\n";
}
sample_hello_world('Ginger Rogers','Ms.');
sample_hello_world('Fred Astaire');
```

此时即可以只向sample_hello_world中传递一个参数，也可以传递完整的两个参数。 那同样的功能我们怎样在扩展函数里实现呢？我们需要借助zend_parse_parameters中的(|)参数，这个参数之前的参数被认为是必须的，之后的便认为是非必须的了,如果没有传递，则不会去修改载体。

```
ZEND_FUNCTION(sample_hello_world) {
    char *name;
    int name_len;
    char *greeting = "Mr./Mrs.";
    int greeting_len = sizeof("Mr./Mrs.") - 1;


    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s|s",
      &name, &name_len, &greeting, &greeting_len) == FAILURE) {
        RETURN_NULL();
    }
    php_printf("Hello ");
    PHPWRITE(greeting, greeting_len);
    php_printf(" ");
    PHPWRITE(name, name_len);
    php_printf("!\n");
}
```

如果你不传递第二个参数，则扩展函数会被认为默认而不去修改载体。所以，我们需要自己来预先设置有载体的值，它往往是是NULL，或者一个与函数逻辑有关的值。 每个zval，包括IS_NULL型的zval，都需要占用一定的内存空间，并且需要cpu的计算资源来为它申请内存、初始化，并在它们完成工作后释放掉。但是很多代码都都没有意识到这一点。有很多代码都会把一个null型的值包裹成zval的IS_NULL类型，在扩展开发里这种操作是可以优化的，我们可以把参数接收城C语言里的NULL。我们就这一个问题看以下代码：

```
ZEND_FUNCTION(sample_arg_fullnull) {
    zval *val;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z",&val) == FAILURE) {
        RETURN_NULL();
    }
    if (Z_TYPE_P(val) == IS_NULL) {
        val = php_sample_make_defaultval(TSRMLS_C);
    }
    ...
}
ZEND_FUNCTION(sample_arg_nullok) {
    zval *val;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "z!",
                                    &val) == FAILURE) {
        RETURN_NULL();
    }
    if (!val) {
        val = php_sample_make_defaultval(TSRMLS_C);
    }
}
```

这两段代码乍看起来并没有什么很大的不同，但是第一段代码确实需要更多的cpu和内存资源。可能这个技巧在平时并没多大用，不过技多不压身，知道总比不知道好。

Forced Separation

当一个变量被传递给函数时候，无论它是否被引用，它的refcoung__gc属性都会加一，至少成为2。一份是它自己，另一份是传递给函数的copy。在改变这个zval之前，有时会需要提前把它分成实际意义上的两份copy。这就是"/"格式符的作用。它将把写时复制的zval提前分成两个完整独立的copy，从而使我们可以在下面的代码中随意的对其进行操作。否则我们可能需要不停的提醒自己对接收的参数进行分离等操作。
Like the NULL flag, this modifier goes after the type it means to impact. Also like the NULL flag, you won't know you need this feature until you actually have a use for it.

zend_get_arguments()

如果你想让你的扩展能够兼容老版本的PHP，或者你只想以zval为载体来接收参数，便可以考虑使用*zend_get_parameters()*函数来接收参数。 *zend_get_parameters()*与*zend_parse_parameters()*不同，从名字上我们便可以看出，它直接获取，而不做解析。首先，它不会自动进行类型转换，所有的参数在扩展实现中的载体都需要是*zval*类型的，下面让我们来看一个最简单的例子：

```
ZEND_FUNCTION(sample_onearg) {
    zval *firstarg;
    if (zend_get_parameters(ZEND_NUM_ARGS(), 1, &firstarg)== FAILURE) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Expected at least 1 parameter.");
```

```
        RETURN_NULL();
    }
    /* Do something with firstarg... */
}
```

其次，zend_get_parameters()在接收失败的时候，并不会自己抛出错误，它也不能方便的处理具有默认值的参数。 最后一点与zend_parse_parameters不同的是，它会自动的把所有符合copy-on-write的zval进行强制分离，生成一个崭新的copy送到函数内部。如果你希望用它其它的特性，而唯独不需要这个功能，可以去尝试一下用zend_get_parameters_ex()函数来接收参数。 为了不对copy-on-write的变量进行分离操作，zend_get_parameters_ex()的参数是zval*类型的，而不是*zval*。 这个函数不太经常用，可能只会在你碰到一些极端问题时候才会想到它，而它用起来却很简单：

```
ZEND_FUNCTION(sample_onearg) {
    zval **firstarg;
    if (zend_get_parameters_ex(1, &firstarg) == FAILURE) {
        WRONG_PARAM_COUNT;
    }
    /* Do something with firstarg... */
}
```

注意zend_get_parameters_ex不需要ZEND_NUM_ARGS()作为参数，因为它是在是在后期加入的，那个参数已经不再需要了。
上面例子中还使用了WRONG_PARAM_COUNT宏,它的功能是抛出一个E_WARNING级别的错误信息，并自动return。

可变参数

有两种其它的zend*get_parameter***函数，专门用来解决参数很多或者无法提前知道参数数目的问题。想一下php语言中var_dump()函数的用法，我们可以向其传递任意数量的参数，它在内核中的实现其实是这样的：

```
ZEND_FUNCTION(var_dump) {
    int i, argc = ZEND_NUM_ARGS();
    zval ***args;

    args = (zval ***)safe_emalloc(argc, sizeof(zval **), 0);
    if (ZEND_NUM_ARGS() == 0 || zend_get_parameters_array_ex(argc, args) == FAILURE) {
        efree(args);
        WRONG_PARAM_COUNT;
    }
    for (i=0; i<argc; i++) {
        php_var_dump(args[i], 1 TSRMLS_CC);
    }
    efree(args);
}
```

程序首先获取参数数量，然后通过safe_emalloc函数申请了相应大小的内存来存放这些zval*类型的参数。这里使用了*zend_get_parameters_array_ex()*函数来把传递给函数的参数填充到*args*中。你可能已经立即想到，还存在一个名为*zend_get_parameters_array()*的函数，唯一不同的是它将*zval*类型的参数填充到args

中，并且需要ZEND_NUM_ARGS()作为参数。

links

- 6.3 第六章小结
- 7.2 Arg Info 与类型绑定

7.2 函数的参数

在前面的章节中我们已经介绍过arg info了，下面我们看一下如何通过其实现类型绑定，但这个特性只能在Zend Engine 2也就是PHP5中使用。 让我们再回顾一下ZE2's argument info结构。每一个arg info结构的声明都是通过ZEND*BEGIN_ARG_INFO()*或者*ZEND_BEGIN_ARG_INFO_EX()*宏函数开始的，然后紧跟着几行*ZEND_ARG**INFO()宏函数，最终以ZEND_END_ARG_INFO()宏函数结束。每个宏的基本作用我们可以在第6章的最后一节看到。如果我们想重写一下PHP语言中的count()函数，可以：

```
ZEND_FUNCTION(sample_count_array)
{
    zval *arr;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "a",&arr) == FAILURE)
    {
        RETURN_NULL();
    }
    RETURN_LONG(zend_hash_num_elements(Z_ARRVAL_P(arr)));
}
```

zend_parse_parameters()本身可以保证传递过来的参数是一个数组。但是如果我们通过zend_get_parameter()函数来接收参数的话就没这么幸运了，需要我们自己进行类型校对。如果想让内核自动完成类型校对，便需要arg_info上场了：

```
    ZEND_BEGIN_ARG_INFO(php_sample_array_arginfo, 0)
        ZEND_ARG_ARRAY_INFO(0, arr, 0)
    ZEND_END_ARG_INFO()

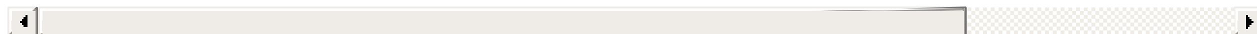
....
    PHP_FE(sample_count_array, php_sample_array_arginfo)
....
```

这样我们便把类型校对的工作交给了Zend Engine，是不是有种如释重负的感觉！You've also given your argument a name so that the generated error messages can be more meaningful to script writers attempting to use your API. 我们同样可以对参数中的对象进行校验，限制其是继承自某个类或者实现了某个接口等等。

```
    ZEND_BEGIN_ARG_INFO(php_sample_class_arginfo, 0)
        ZEND_ARG_OBJ_INFO(1, obj, stdClass, 0)
    ZEND_END_ARG_INFO()
```

需要注意的是，此时第一个参数的值是数字1，代表着以引用的方式传递。其实这个参数对于对象来说几乎没用，因为ZE2中所有的对象在当作函数参数的时候都是默认以引用的形式传递的。但是我们又必须把这个参数设置为数字1，除非你不想让你的扩展与PHP4兼容。在PHP4中，对象是传递的一个完整Copy，而非通过引用。

```
<div class="tip-common">对于数组和对象参数，不要忘记最后的允许为NULL的参数。更多的信息请参考第6章最后
```



通过arg info的方式来实现类型绑定的功能只对ZE2有效，也就是PHP5+。如果你想在PHP4上实现相应的功能，那需要用 zend_get_parameters()函数来接收参数，然后通过Z_TYPE_P()宏函数来检测参数的类型或者通过convert_to_type()函数进行类型转换。

links

- 7.1 zend_parse_parameters
- 7.3 第七章小结

7.3 函数的参数

现在我们已经可以编写一个更真实的函数了，既可以接收用户传递过来的参数，也可以返回数据给调用者。为了写出高质量的代码，还需要我们多花点心思在zval的写时复制等特殊机制上，否则便会在接收参数和返回数据时留下一些bug。 下面的章节里，让我们去看一下PHP语言里强大的数组类型是如何在内核中实现的，去探究内核中的HashTable结构，从而能编写出更强大的PHP扩展。

links

- 7.2 Arg Info 与类型绑定
- 8 使用HashTable与{数组}

8 使用HashTable与{数组}

在C语言中，我们可以自定义各种各样的数据结构，用来把很多数据保存在一个变量里面，但是每种数据结构都有自己的优缺点，PHP内核规模如此庞大，是否已经找到了一些非常棒的解决方法呢？

links

- 7.3 第七章小结
- 8.1 数组(C中的)与链表

8.1 使用HashTable与{数组}

我们在评选各种数据结构时，往往会考虑我们需要处理的数据规模以及需要的性能。下面让我们简要的看一下看C语言中数组和链表的一些事情。

数组

作者这里用的不是Array，而是Vector，可能指的是C++里的Vector， 它与数组几乎是完全一样的，唯一的不同便是可以实现动态存储。 本节下文都是用数组一词代替之， 请各位注意。数组是内存中一块连续的区域，其每一个元素都具有一个唯一的下标值。

```
int a[3];
a[0]=1;
a[2]=3;
```

不仅是整数，其它类型的变量也可以保存在数组中，比如我们上一章用到的zend_get_parameters_array_ex()，便把很多zval**类型的变量保存到一个数组里， 为了使其正常工作， 我们提前向系统申请了相应大小的内存空间。

```
zval ***args = safe_emalloc(ZEND_NUM_ARGS(), sizeof(zval**), 0);
```

这里我们仍然可以用一个整数来当作下标去数组中取出我们想要的数据，就像var_dump()的实现中通过args[i]来获取参数并把它传递给php_var_dump()函数那样。 使用数组最大的好处便是速度！读写都可以在O(1)内完成，因为它每个元素的大小都是一致的，只要知道下标，便可以瞬间计算出其对应的元素在内存中的位置，从而直接取出或者写入。

链表

链表也是一种经常被使用的一种数据结构。链表中的每一个元素都至少有两个元素，一个指向它的下一个元素，一个用来存放它自己的数据，就像下面定义的那样：

```
typedef struct _namelist namelist;
struct _namelist
{
    struct _namelist *next;
    char *name;
};
```

我们可以声明一个其类型的元素：

```
static namelist *people;
```

假设每一个元素都代表一个人，元素中的name属性便是这个人的名字，我们通过这样的语句来得到它：

people->name; 第二个属性指向后面的一个元素，那我们便可以这样来访问下一个人的名字：people->next->name, 或者下一个人的下一个人的名字：people->next->next->name,一次类推，直到next的值是NULL，代表结束。

```
//通过一个循环来遍历这个链表中的所有人~
void name_show(namelist *p)
{
    while (p)
    {
        printf("Name: %s\n", p->name);
        p = p->next;
    }
}
```

链表可以被用来实现FIFO模式，达到先进者先出的目的！

```
static namelist *people = NULL, *last_person = NULL;
void name_add(namelist *person)
{
    person->next = NULL;
    if (!last_person) {
        /* No one in the list yet */
        people = last_person = person;
        return;
    }
    /* Append new person to the end of the list */
    last_person->next = person;

    /* Update the list tail */
    last_person = person;
}
namelist *name_pop(void)
{
    namelist *first_person = people;
    if (people) {
        people = people->next;
    }
    return first_person;
}
```

这样，我们便可以随意的向这个链表中添加或者删除数据，而不像数组那样，谨慎的考虑是否越界等问题。 上面实现的结构的学名叫做单向链表，也有地方叫单链表，反正是比较简单的意思~。它有一个致命的缺点，就是我们在插入或者读取某条数据的时候，都需要从这个链表的开始，一个个元素的向下寻找，直到找到这个元素为止。如果链表中的元素比较多，那它很容易成为我们程序中的CPU消耗大户，进而引起性能问题。为了解决这个问题，先人们发明了双向链表：

```
typedef struct _namelist namelist;
struct
{
    namelist *next, *prev;
    char *name;
} _namelist;
```

改动其实不大，就是在每个元素中都添加了一个prev属性，用来指向它的上一个元素。

```
void name_add(namelist *person)
{
    person->next = NULL;
    if (!last_person)
    {
        /* No one in the list yet */
        people = last_person = person;
        person->prev = NULL;
        return;
    }
    /* Append new person to the end of the list */
    last_person ->next = person;
    person->prev = last_person;

    /* Update the list tail */
    last_person = person;
}
```

单单通过上面的程序你还体会不到它的好处，但是设想一下，如果现在你有这个链表中其中一个元素的地址，并且想把它从链表中删除，那我们该怎么做呢？如果是单向链表的话，我们只能这样做：

```
void name_remove(namelist *person)
{
    namelist *p;
    if (person == people) {
        /* Happens to be the first person in the list */
        people = person->next;
        if (last_person == person) {
            /* Also happens to be the last person */
            last_person = NULL;
        }
        return;
    }
    /* Search for prior person */
    p = people;
    while (p) {
        if (p->next == person) {
            /* unlink */
            p->next = person->next;
            if (last_person == person) {
                /* This was the last element */
                last_person = p;
            }
            return;
        }
        p = p->next;
    }
    /* Not found in list */
}
```

现在让我们来看看双向链表是怎样来处理这个问题的：

```
void name_remove(namelist *person)
{
    if (people == person) {
        people = person->next;
    }
    if (last_person == person) {
        last_person = person->prev;
    }
    if (person->prev) {

        person->prev->next = person->next;
    }
    if (person->next) {
        person->next->prev = person->prev;
    }
}
```

对元素的遍历查找不见了，取而代之的是一个O(1)的运算，这将极大的提升我们程序的性能。

王者归来：HashTable才是我们的银蛋！

也许你已经非常喜欢使用数组或者链表了，但我还是要向你推荐一种威力极大的数据结构，有了它之后，你可能会立即抛弃前两者，它就是HashTable. HashTable既具有双向链表的优点，同时具有能与数据匹敌的操作性能，这个数据结构几乎是PHP内核的实现的基础，我们在内核代码的任何地方都发现它的痕迹。

第二章我们接触过，所有的用户端定义的变量保存在一个符号表里，而这个符号表其实就是一个HashTable，它的每一个元素都是一个zval*类型的变量。不仅如此，保存用户定义的函数、类、资源等的容器都是以HashTable的形式在内核中实现的。 Zend Engine中HashTable的元素其实是指针，对其的这个改进使得HashTable能够包容各种类型的数据，从小小的标量，到复杂的PHP5中实现的类等复合数据。本章接下来的内容，我们将详细的研究如何使用zend内置的API来操作HashTable这个数据结构。

links

- 8 使用HashTable与{数组}
- 8.2 操作HashTable的API

8.2 使用HashTable与{数组}

Zend把与HashTable有关的API分成了好几类以便于我们寻找，这些API的返回值大多都是常量SUCCESS或者FAILURE。

创建HashTable

> 下面在介绍函数原型的时候都使用了ht名称，但是我们在编写扩展的时候，一定不要使用这个名称，因为一些PHP宏展开后会声明这个名称的变量，进而引发命名冲突。

创建并初始化一个HashTable非常简单，只要使用zend_hash_init函数即可，它的定义如下：

```
int zend_hash_init(
    HashTable *ht,
    uint nSize,
    hash_func_t pHashFunction,
    dtor_func_t pDestructor,
    zend_bool persistent
);
```

- ***ht**是指针，指向一个HashTable，我们既可以&一个已存在的HashTable变量，也可以通过emalloc()、pemalloc()等函数来直接申请一块内存，不过最常用的方法还是用ALLOC_HASHTABLE(ht)宏来让内核自动的替我们完成这项工作。ALLOC_HASHTABLE(ht)所做的工作相当于ht = emalloc(sizeof(HashTable));
- **nSize**代表着这个HashTable可以拥有的元素的最大数量(HashTable能够包含任意数量的元素，这个值只是为了提前申请好内存，提高性能，省的不停的进行rehash操作)。在我们添加新的元素时，这个值会根据情况决定是否自动增长，有趣的是，这个值永远都是2的次方，如果你给它的值不是一个2的次方的形式，那它将自动调整成大于它的最小的2的次方值。它的计算方法就像这样：nSize = pow(2, ceil(log(nSize, 2)));
- **pHashFunction**是早期的Zend Engine中的一个参数，为了兼容没有去掉它，但它已经没有用处了，所以我们直接赋成NULL就可以了。在原来，它其实是一个钩子，用来让用户自己hook一个散列函数，替换php默认的DJBX33A算法实现。
- **pDestructor**也代表着一个回调函数，当我们删除或者修改HashTable中其中一个元素时候便会调用，它的函数原型必须是这样的：void method_name(void *pElement);*这里的pElement是一个指针，指向HashTable中那么将要被删除或者修改的那个数据，而数据的类型往往也是个指针。
- **persistent**是最后一个参数，它的含义非常简单。如果它为true，那么这个HashTable将永远存在于内存中，而不会在RSHUTDOWN阶段自动被注销掉。此时第一个参数ht所指向的地址必须是通过pemalloc()函数申请的。

举个例子，PHP内核在每个Request请求的头部都调用了这个函数来初始化symbol_table。

```
zend_hash_init(&EG(symbol_table), 50, NULL, ZVAL_PTR_DTOR, 0);

//#define ZVAL_PTR_DTOR (void (*)(void *)) zval_ptr_dtor_wrapper
```

如你所见，每个元素在从符号表里删除的时候(比如执行"<?php unset($foo);"操作)，都会触发ZVAL_PTR_DTOR宏代表的函数来对其进行与引用计数有关的操作。 因为50不是2的整数幂形式，所以它会在函数执行时被调成成64。

添加&&修改

我们有四个常用的函数来完成这项操作，它们的原型分别如下：

```
int zend_hash_add(
    HashTable *ht,          //待操作的ht
    char *arKey,              //索引，如"my_key"
    uint nKeyLen,         //字符串索引的长度，如6
    void **pData,         //要插入的数据，注意它是void **类型的。int *p,i=1;p=&i,pData=&p;。
    uint nDataSize,
    void *pDest             //如果操作成功，则pDest=*pData;
);

int zend_hash_update(
    HashTable *ht,
    char *arKey,
    uint nKeyLen,
    void *pData,
    uint nDataSize,
    void **pDest
);

int zend_hash_index_update(
    HashTable *ht,
    ulong h,
    void *pData,
    uint nDataSize,
    void **pDest
);

int zend_hash_next_index_insert(
    HashTable *ht,
    void *pData,
    uint nDataSize,
    void **pDest
);
```

前两个函数用户添加带字符串索引的数据到HashTable中，就像我们在PHP中使用的那样:$foo['bar'] = 'baz';用C来完成便是：

```
zend_hash_add(fooHashTbl, "bar", sizeof("bar"), &barZval, sizeof(zval*), NULL);
```

zend_hash_add()和zend_hash_update()唯一的区别就是如果这个key已经存在了，那么zend_hash_add()将返回FAILURE，而不会修改原有数据。 接下来的两个函数用于像HT中添加数字索引的数据，zend_hash_next_index_insert()函数则不需要索引值参数，而是自己直接计算出下一个数字索引值. 但是如果我们想获取下一个元素的数字索引值，也是有办法的，可以使用zend_hash_next_free_element()函数：

```
ulong nextid = zend_hash_next_free_element(ht);
zend_hash_index_update(ht, nextid, &data, sizeof(data), NULL);
```

所有这些函数中，如果pDest不为NULL，内核便会修改其值为被操作的那个元素的地址。在下面的代码中这个参数也有同样的功能。

查找

因为HashTable中有两种类型的索引值，所以需要两个函数来执行find操作。

```
int zend_hash_find(HashTable *ht, char *arKey, uint nKeyLength,void **pData);
int zend_hash_index_find(HashTable *ht, ulong h, void **pData);
```

第一种就是我们处理PHP语言中字符串索引数组时使用的，第二种是我们处理PHP语言中数字索引数组使用的。Recall from Chapter 2 that when data is added to a HashTable, a new memory block is allocated for it and the data passed in is copied; when the data is extracted back out it is the pointer to that data which is returned. The following code fragment adds data1 to the HashTable, and then extracts it back out such that at the end of the routine, *data2 contains the same contents as* data1 even though the pointers refer to different memory addresses.

```
void hash_sample(HashTable *ht, sample_data *data1)
{
    sample_data *data2;
    ulong targetID = zend_hash_next_free_element(ht);
    if (zend_hash_index_update(ht, targetID,
            data1, sizeof(sample_data), NULL) == FAILURE) {
            /* Should never happen */
            return;
    }
    if(zend_hash_index_find(ht, targetID, (void **)&data2) == FAILURE) {
        /* Very unlikely since we just added this element */
        return;
    }
    /* data1 != data2, however *data1 == *data2 */
}
```

除了读取，我们还需要检测某个key是否存在：

```
int zend_hash_exists(HashTable *ht, char *arKey, uint nKeyLen);
int zend_hash_index_exists(HashTable *ht, ulong h);
```

这两个函数返回SUCCESS或者FAILURE，分别代表着是否存在：

```
if( zend_hash_exists(EG(active_symbol_table),"foo", sizeof("foo")) == SUCCESS )
{
    /* $foo is set */
}
else
```

```
{
    /* $foo does not exist */
}
```

提速!

```
ulong zend_get_hash_value(char *arKey, uint nKeyLen);
```

当我们需要对同一个字符串的key进行许多操作时候，比如先检测有没，然后插入，然后修改等等，这时我们便可以使用zend_get_hash_value函数来对我们的操作进行加速！这个函数的返回值可以和quick系列函数使用，达到加速的目的(就是不再重复计算这个字符串的散列值，而直接使用已准备好的)！

```
int zend_hash_quick_add(
    HashTable *ht,
    char *arKey,
    uint nKeyLen,
    ulong hashval,
    void *pData,
    uint nDataSize,
    void **pDest
);

int zend_hash_quick_update(
    HashTable *ht,
    char *arKey,
    uint nKeyLen,
    ulong hashval,
    void *pData,
    uint nDataSize,
    void **pDest
);

int zend_hash_quick_find(
    HashTable *ht,
    char *arKey,
    uint nKeyLen,
    ulong hashval,
    void **pData
);

int zend_hash_quick_exists(
    HashTable *ht,
    char *arKey,
    uint nKeyLen,
    ulong hashval
);
```

虽然很意外，但你还是要接受没有zend_hash_quick_del()这个函数。quick类函数会在下面这种场合中用到：

```
void php_sample_hash_copy(HashTable *hta, HashTable *htb,char *arKey, uint nKeyLen TSRMLS
{
```

```
    ulong hashval = zend_get_hash_value(arKey, nKeyLen);
    zval **copyval;

    if (zend_hash_quick_find(hta, arKey, nKeyLen,hashval, (void**)&copyval) == FAILURE)
    {
        //标明不存在这个索引
        return;
    }

    //这个zval已经被其它的Hashtable使用了，这里我们进行引用计数操作。
    (*copyval)->refcount__gc++;
    zend_hash_quick_update(htb, arKey, nKeyLen, hashval,copyval, sizeof(zval*), NULL);
}
```

复制与合并(Copy And Merge)

在PHP语言中，我们经常需要进行数组间的Copy与Merge操作，所以php语言中的数组在C语言中的实现HashTable也肯定会经常碰到这种情况。为了简化这一类操作，内核中早已准备好了相应的API供我们使用。

```
void zend_hash_copy(
    HashTable *target,
    HashTable *source,
    copy_ctor_func_t pCopyConstructor,
    void *tmp,
    uint size
);
```

- *source中的所有元素都会通过pCopyConstructor函数Copy到*target中去，我们还是以PHP语言中的数组举例，pCopyConstructor这个hook使得我们可以在copy变量的时候对他们的ref_count进行加一操作。target中原有的与source中索引位置的数据会被替换掉，而其它的元素则会被保留，原封不动。
- tmp参数是为了兼容PHP4.0.3以前版本的，现在赋值为NULL即可。
- size参数代表每个元素的大小，对于PHP语言中的数组来说，这里的便是sizeof(zval*)了。

```
void zend_hash_merge(
    HashTable *target,
    HashTable *source,
    copy_ctor_func_t pCopyConstructor,
    void *tmp,
    uint size,
    int overwrite
);
```

zend_hash_merge()与zend_hash_copy唯一的不同便是多了个int类型的overwrite参数，当其值非0的时候，两个函数的工作是完全一样的；如果overwrite参数为0，则zend_hash_merge函数就不会对target中已有索引的值进行替换了。

```
typedef zend_bool (*merge_checker_func_t)(HashTable *target_ht,void *source_data, zend_ha
void zend_hash_merge_ex(
```

```
    HashTable *target,
    HashTable *source,
    copy_ctor_func_t pCopyConstructor,
    uint size,
    merge_checker_func_t pMergeSource,
    void *pParam
);
```

这个函数又繁琐了些，与zend_hash_copy相比，其多了两个参数，多出来的pMergeSoure回调函数允许我们选择性的进行merge，而不是全都merge。The final form of this group of functions allows for selective copying using a merge checker function. The following example shows zend_hash_merge_ex() in use to copy only the associatively indexed members of the source HashTable (which happens to be a userspace variable array):

```
zend_bool associative_only(HashTable *ht, void *pData,zend_hash_key *hash_key, void *pPar
{
    //如果是字符串索引
    return (hash_key->arKey && hash_key->nKeyLength);
}

void merge_associative(HashTable *target, HashTable *source)
{
    zend_hash_merge_ex(target, source, zval_add_ref,sizeof(zval*), associative_only, NULL
}
```

遍历

在PHP语言中，我们有很多方法来遍历一个数组，对于数组的本质HashTable，我们也有很多办法来对其进行遍历操作。首先最简单的一种办法便是使用一种与PHP语言中forech语句功能类似的函数——zend_hash_apply，它接收一个回调函数，并将HashTable的每一个元素都传递给它。

```
typedef int (*apply_func_t)(void *pDest TSRMLS_DC);
void zend_hash_apply(HashTable *ht,apply_func_t apply_func TSRMLS_DC);
```

下面是另外一种遍历函数：

```
typedef int (*apply_func_arg_t)(void *pDest,void *argument TSRMLS_DC);
void zend_hash_apply_with_argument(HashTable *ht,apply_func_arg_t apply_func, void *data
```

通过上面的函数可以在执行遍历时向回调函数传递任意数量的值，这在一些diy操作中非常有用。

上述函数对传给它们的回调函数的返回值有一个共同的约定，详细介绍下下表：

```
表格 8.1. 回调函数的返回值
Constant                        Meaning
```

```
ZEND_HASH_APPLY_KEEP          结束当前请求，进入下一个循环。与PHP语言foreach语句中的一次循环执行完毕或
ZEND_HASH_APPLY_STOP          跳出，与PHP语言foreach语句中的break关键字的作用一样。
ZEND_HASH_APPLY_REMOVE          删除当前的元素，然后继续处理下一个。相当于在PHP语言中：unset($foo[$k
```

我们来一下PHP语言中的forech循环：

```
<?php
foreach($arr as $val) {
    echo "The value is: $val\n";
}
?>
```

那我们的回调函数在C语言中应该这样写：

```
int php_sample_print_zval(zval **val TSRMLS_DC)
{
    //重新copy一个zval，防止破坏原数据
    zval tmpcopy = **val;
    zval_copy_ctor(&tmpcopy);

    //转换为字符串
    INIT_PZVAL(&tmpcopy);
    convert_to_string(&tmpcopy);

       //开始输出
    php_printf("The value is: ");
    PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
    php_printf("\n");

    //毁尸灭迹
    zval_dtor(&tmpcopy);

       //返回，继续遍历下一个～
    return ZEND_HASH_APPLY_KEEP;
}
```

遍历我们的HashTable：

```
//生成一个名为arrht、元素为zval*类型的HashTable
zend_hash_apply(arrht, php_sample_print_zval TSRMLS_CC);
```

> 再次提醒，保存在HashTable中的元素并不是真正的最终变量，而是指向它的一个指针。我们的上面的遍历函数接收的是一个zval*类型的参数。 ```c typedef int *(apply_func_args_t)(void pDest,int num_args, va_list args, zend_hash_key* hash_key); void zend_hash_apply_with_arguments(HashTable *ht,apply_func_args_t apply_func, int numargs, ...);

```
为了能在遍历时同时接收索引的值，我们必须使用第三种形式的zend_hash_apply！就像PHP语言中这样的功能：
```php
```
```

```
<?php
foreach($arr as $key => $val)
{
    echo "The value of $key is: $val\n";
}
?>
```

为了配合zend_hash_apply_with_arguments()函数，我们需要对我们的遍历执行函数做一下小小的改动，使其接受索引作为一个参数：

```
int php_sample_print_zval_and_key(zval **val,int num_args,va_list args,zend_hash_key *has
{
    //重新copy一个zval，防止破坏原数据
    zval tmpcopy = **val;
    /* tsrm_ls is needed by output functions */
    TSRMLS_FETCH();
    zval_copy_ctor(&tmpcopy);
    INIT_PZVAL(&tmpcopy);

    //转换为字符串
    convert_to_string(&tmpcopy);

    //执行输出
    php_printf("The value of ");
    if (hash_key->nKeyLength)
    {
        //如果是字符串类型的key
        PHPWRITE(hash_key->arKey, hash_key->nKeyLength);
    }
    else
    {
        //如果是数字类型的key
        php_printf("%ld", hash_key->h);
    }

    php_printf(" is: ");
    PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
    php_printf("\n");

    //毁尸灭迹
    zval_dtor(&tmpcopy);
    /* continue; */
    return ZEND_HASH_APPLY_KEEP;
}
```

执行遍历：

```
zend_hash_apply_with_arguments(arrht,php_sample_print_zval_and_key, 0);
```

这个函数通过C语言中的可变参数特性来接收参数。This particular example required no arguments to be passed; for information on extracting variable argument lists from va_list args, see the POSIX documentation pages for va_start(), va_arg(), and va_end().

当我们检查这个hash_key是字符串类型还是数字类型时，是通过nKeyLength属性来检测的,而不是arKey属性。这是因为内核有时候会留在arKey属性里些脏数据，但nKeyLength属性是安全的，可以安全的使用。甚至对于空字符串索引，它也照样能处理。比如：$foo[''] ="Bar";索引的值是NULL字符，但它的长度却是包括最后这个NULL字符的，所以为1。

向前遍历HashTable

有时我们希望不用回调函数也能遍历一个数组的数据，为了实现这个功能，内核特意的为每个HashTable加了个属性：The internal pointer（内部指针）。

```
<?php
    $arr = array('a'=>1, 'b'=>2, 'c'=>3);
    reset($arr);
    while (list($key, $val) = each($arr)) {
        /* Do something with $key and $val */
    }
    reset($arr);
    $firstkey = key($arr);
    $firstval = current($arr);
    $bval = next($arr);
    $cval = next($arr);
?>
```

ZEND内核中有一组操作HashTable的功能与以上函数功能类似的函数：

```
/* reset() */
void zend_hash_internal_pointer_reset(HashTable *ht);

/* key() */
int zend_hash_get_current_key(HashTable *ht,char **strIdx, unit *strIdxLen,ulong *numIdx,

/* current() */
int zend_hash_get_current_data(HashTable *ht, void **pData);

/* next()/each() */
int zend_hash_move_forward(HashTable *ht);

/* prev() */
int zend_hash_move_backwards(HashTable *ht);

/* end() */
void zend_hash_internal_pointer_end(HashTable *ht);

/* 其他的...... */
int zend_hash_get_current_key_type(HashTable *ht);
int zend_hash_has_more_elements(HashTable *ht);
```

```
<div class="tip-common">PHP语言中的next()、prev()、end()函数在移动完指针之后，都通过调用zend_has
```

现在我们用另外一种方法来实现上面的forech：

```
void php_sample_print_var_hash(HashTable *arrht)
{

    for(
        zend_hash_internal_pointer_reset(arrht);
        zend_hash_has_more_elements(arrht) == SUCCESS;
        zend_hash_move_forward(arrht))
    {
        char *key;
        uint keylen;
        ulong idx;
        int type;
        zval **ppzval, tmpcopy;

        type = zend_hash_get_current_key_ex(arrht, &key, &keylen,&idx, 0, NULL);
        if (zend_hash_get_current_data(arrht, (void**)&ppzval) == FAILURE)
        {
            /* Should never actually fail
             * since the key is known to exist. */
            continue;
        }

        //重新copy一个zval，防止破坏原数据
        tmpcopy = **ppzval;
        zval_copy_ctor(&tmpcopy);
        INIT_PZVAL(&tmpcopy);

        convert_to_string(&tmpcopy);

        /* Output */
        php_printf("The value of ");
        if (type == HASH_KEY_IS_STRING)
        {
            /* String Key / Associative */
            PHPWRITE(key, keylen);
        } else {
            /* Numeric Key */
            php_printf("%ld", idx);
        }
        php_printf(" is: ");
        PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
        php_printf("\n");
        /* Toss out old copy */
        zval_dtor(&tmpcopy);
    }
}
```

上面的代码你应该都能看懂了，唯一还没接触到的可能是zend_hash_get_current_key()函数的返回值，它的返回值见表8.2。

```
Constant                            Meaning

HASH_KEY_IS_STRING                当前元素的索引是字符串类型的。therefore, a pointer to the el
```

```
HASH_KEY_IS_LONG                当前元素的索引是数字型的。
HASH_KEY_NON_EXISTANT            HashTable中的内部指针已经移动到尾部，不指向任何元素。
```

Preserving the Internal Pointer

在我们遍历一个HashTable时，一般是很难陷入死循环的。When iterating through a HashTable, particularly one containing userspace variables, it's not uncommon to encounter circular references, or at least self-overlapping loops. If one iteration context starts looping through a HashTable and the internal pointer reachesfor examplethe halfway mark, a subordinate iterator starts looping through the same HashTable and would obliterate the current internal pointer position, leaving the HashTable at the end when it arrived back at the first loop.

The way this is resolvedboth within the zend_hash_apply implementation and within custom move forward usesis to supply an external pointer in the form of a HashPosition variable.

Each of the zend*hash() functions listed previously has a zendhash*_ex() counterpart that accepts one additional parameter in the form of a pointer to a HashPostion data type. Because the HashPosition variable is seldom used outside of a short-lived iteration loop, it's sufficient to declare it as an immediate variable. You can then dereference it on usage such as in the following variation on the php_sample_print_var_hash() function you saw earlier:

```
void php_sample_print_var_hash(HashTable *arrht)
{
    HashPosition pos;
    for(zend_hash_internal_pointer_reset_ex(arrht, &pos);
    zend_hash_has_more_elements_ex(arrht, &pos) == SUCCESS;
    zend_hash_move_forward_ex(arrht, &pos)) {
        char *key;
        uint keylen;
        ulong idx;
        int type;

        zval **ppzval, tmpcopy;

        type = zend_hash_get_current_key_ex(arrht,
                                &key, &keylen,
                                &idx, 0, &pos);
        if (zend_hash_get_current_data_ex(arrht,
                    (void**)&ppzval, &pos) == FAILURE) {
            /* Should never actually fail
             * since the key is known to exist. */
            continue;
        }
        /* Duplicate the zval so that
         * the original's contents are not destroyed */
        tmpcopy = **ppzval;
        zval_copy_ctor(&tmpcopy);
        /* Reset refcount & Convert */
        INIT_PZVAL(&tmpcopy);
        convert_to_string(&tmpcopy);
        /* Output */
        php_printf("The value of ");
```

```
        if (type == HASH_KEY_IS_STRING) {
            /* String Key / Associative */
            PHPWRITE(key, keylen);
        } else {
            /* Numeric Key */
            php_printf("%ld", idx);
        }
        php_printf(" is: ");
        PHPWRITE(Z_STRVAL(tmpcopy), Z_STRLEN(tmpcopy));
        php_printf("\n");
        /* Toss out old copy */
        zval_dtor(&tmpcopy);
    }
}
With these very slight additions, the HashTable's true internal pointer is preserved in w
### 删除
内核中一共预置了四个删除HashTable元素的函数，头两个是用户删除某个确定索引的数据：
<code c>
int zend_hash_del(HashTable *ht, char *arKey, uint nKeyLen);
int zend_hash_index_del(HashTable *ht, ulong h);
```

它们两个分别用来删除字符串索引和数字索引的数据，操作完成后都返回SUCCESS或者FAILURE表示成功or失败。 回顾一下最上面的叙述，当一个元素被删除时，会激活HashTable的destructor回调函数。

```
void zend_hash_clean(HashTable *ht);
void zend_hash_destroy(HashTable *ht);
```

前者用于将HashTable中的元素全部删除，而后者是将这个HashTable自身也毁灭掉。 现在让我们来完整的回顾一下HashTable的创建、添加、删除操作。

```
int sample_strvec_handler(int argc, char **argv TSRMLS_DC)
{
    HashTable *ht;

    //分配内存
    ALLOC_HASHTABLE(ht);

    //初始化
    if (zend_hash_init(ht, argc, NULL,ZVAL_PTR_DTOR, 0) == FAILURE) {
        FREE_HASHTABLE(ht);
        return FAILURE;
    }

    //填充数据
    while (argc) {
        zval *value;
        MAKE_STD_ZVAL(value);
        ZVAL_STRING(value, argv[argc], 1);
        argv++;
        if (zend_hash_next_index_insert(ht, (void**)&value,
                            sizeof(zval*)) == FAILURE) {
            /* Silently skip failed additions */
            zval_ptr_dtor(&value);
        }
```

```
    }

    //完成工作
    process_hashtable(ht);

    //毁尸灭迹
    zend_hash_destroy(ht);

    //释放ht 为什么不在destroy里free呢，求解释！
    FREE_HASHTABLE(ht);
    return SUCCESS;
}
```

排序、比较and Going to the Extreme(s)

针对HashTable操作的Zend Api中有很多都需要回调函数。首先让我们来处理一下对HashTable中元素大小比较的问题：

```
typedef int (*compare_func_t)(void *a, void *b TSRMLS_DC);
```

这很像PHP语言中usort函数需要的参数，它将比较两个值*a*与b，如果*a*>b,则返回1，相等则返回0，否则返回-1。下面是zend_hash_minmax函数的声明，它就需要我们上面声明的那个类型的函数作为回调函数：int zend_hash_minmax(HashTable *ht, compare_func_t compar,int flag, void* *pData TSRMLS_DC); 这个函数的功能我们从它的名称中便能肯定，它用来比较HashTable中的元素大小。如果flag==0则返回最小值，否则返回最大值！

下面让我们来利用这个函数来对用户端定义的所有函数根据函数名找到最大值与最小值(大小写不敏感～)。

```
//先定义一个比较函数，作为zend_hash_minmax的回调函数。
int fname_compare(zend_function *a, zend_function *b TSRMLS_DC)
{
    return strcasecmp(a->common.function_name, b->common.function_name);
}

void php_sample_funcname_sort(TSRMLS_D)
{
    zend_function *fe;
    if (zend_hash_minmax(EG(function_table), fname_compare,0, (void **)&fe) == SUCCESS)
    {
        php_printf("Min function: %s\n", fe->common.function_name);
    }
    if (zend_hash_minmax(EG(function_table), fname_compare,1, (void **)&fe) == SUCCESS)
    {
        php_printf("Max function: %s\n", fe->common.function_name);
    }
}
```

zend_hash_compare()也许要回调函数，它的功能是将HashTable看作一个整体与另一个HashTable做比较，如果前者大于后者返回1，相等返回0，否则返回-1。

int zend_hash_compare(HashTable *hta, HashTable* htb,compare_func_t compar, zend_bool ordered TSRMLS_DC);

默认情况下它往往是先判断各个HashTable元素的个数，个数多的最大！ 如果两者的元素一样多，然后就比较它们各自的第一个元素，If the ordered flag is set, it compares keys/indices with the first element of htb string keys are compared first on length, and then on binary sequence using memcmp(). If the keys are equal, the value of the element is compared with the first element of htb using the comparison callback function. If the ordered flag is not set, the data portion of the first element of hta is compared against the element with a matching key/index in htb using the comparison callback function. If no matching element can be found for htb, then hta is considered greater than htb and 1 is returned. If at the end of a given loop, hta and htb are still considered equal, comparison continues with the next element of hta until a difference is found or all elements have been exhausted, in which case 0 is returned. 另外一个重要的需要回调函数的API便是排序函数，它需要的回调函数形式是这样的：

```
typedef void (*sort_func_t)(void **Buckets, size_t numBuckets,size_t sizBucket, compare_f
```

This callback will be triggered once, and receive a vector of all the Buckets (elements) in the HashTable as a series of pointers. These Buckets may be swapped around within the vector according to the sort function's own logic with or without the use of the comparison callback. In practice, sizBucket will always be sizeof(Bucket*).

Unless you plan on implementing your own alternative bubblesort method, you won't need to implement a sort function yourself. A predefined sort methodzend_qsortalready exists for use as a callback to zend_hash_sort() leaving you to implement the comparison function only.

```
int zend_hash_sort(HashTable *ht, sort_func_t sort_func,compare_func_t compare_func, int
```

最后一个参数如果为TRUE，则会抛弃HashTable中原有的索引-键关系，将对排列好的新值赋予新的数字键值。PHP语言中的sort函数实现如下：

```
zend_hash_sort(target_hash, zend_qsort,array_data_compare, 1 TSRMLS_CC);
```

array_data_compare是一个返回compare_func_t类型数据的函数，它将按照HashTable中zval*值的大小进行排序。

links

- 8.1 数组(C中的)与链表
- 8.3 在内核中操作PHP语言中数组

8.3 使用HashTable与{数组}

当你在扩展中使用HashTable时候，95%是要存储用户端的变量，就像PHP语言中数组那样。为此，内核中已经准备好了相应的工具，来让我们更加的方便的操作HashTable存储zval*，也就是PHP语言中的数组，即IS_ARRAY常量代表的zval，以下用{数组}来代替PHP语言中的数组这个词。

创建{数组}

创建HashTable有些繁琐，虽然有辅助的宏但还是不能一步完成，而创建{数组}便简单多了，直接使用array_init(zval *arrval*)函数即可，注意它的参数是*zval*类型的！这样，我们像用户端返回数组便简单多了：

```
ZEND_FUNCTION(sample_array)
{
    array_init(return_value);
}

//return_value是zval*类型的，所以我们直接对它调用array_init()函数即可，即把它初始化成了一个空数组。
```

增！

将{数组}初始化后，接下来就要向其添加元素了。因为PHP语言中有多种类型的变量，所以也对应的有多种类型的add*assoc()*、*addindex*、add*next_index**()函数。如：

```
array_init(arrval);

add_assoc_long(zval *arrval, char *key, long lval);
add_index_long(zval *arrval, ulong idx, long lval);
add_next_index_long(zval *arrval, long lval);
```

这三个函数的第一个参数都要被操作的{数组}指针，然后是索引值，最后是变量，唯一不同的是add_next_index_long()函数的索引值是其自己计算出来的。根据上一节的内容我们可以知道，这三个函数分别在内部使用了zend_hash_update()、zend_hash_index_update()与zend_hash_next_index_insert函数。

```
//add_assoc_*系列函数：
add_assoc_null(zval *aval, char *key);
add_assoc_bool(zval *aval, char *key, zend_bool bval);
add_assoc_long(zval *aval, char *key, long lval);
add_assoc_double(zval *aval, char *key, double dval);
add_assoc_string(zval *aval, char *key, char *strval, int dup);
add_assoc_stringl(zval *aval, char *key,char *strval, uint strlen, int dup);
add_assoc_zval(zval *aval, char *key, zval *value);

//备注：其实这些函数都是宏，都是对add_assoc_*_ex函数的封装。

//add_index_*系列函数：
ZEND_API int add_index_long         (zval *arg, ulong idx, long n);
```

```
ZEND_API int add_index_null        (zval *arg, ulong idx            );
ZEND_API int add_index_bool        (zval *arg, ulong idx, int b     );
ZEND_API int add_index_resource    (zval *arg, ulong idx, int r     );
ZEND_API int add_index_double    (zval *arg, ulong idx, double d);
ZEND_API int add_index_string    (zval *arg, ulong idx, const char *str, int duplicate);
ZEND_API int add_index_stringl    (zval *arg, ulong idx, const char *str, uint length, in
ZEND_API int add_index_zval        (zval *arg, ulong index, zval *value);

//add_next_index_long函数：
ZEND_API int add_next_index_long        (zval *arg, long n    );
ZEND_API int add_next_index_null        (zval *arg            );
ZEND_API int add_next_index_bool        (zval *arg, int b    );
ZEND_API int add_next_index_resource    (zval *arg, int r    );
ZEND_API int add_next_index_double        (zval *arg, double d);
ZEND_API int add_next_index_string        (zval *arg, const char *str, int duplicate);
ZEND_API int add_next_index_stringl        (zval *arg, const char *str, uint length, int
ZEND_API int add_next_index_zval        (zval *arg, zval *value);
```

每组函数最后的一个，即zend..._zval()函数，允许我们向这个{数组}中添加资源、对象、{数组}等复合类型的PHP变量。下面让我们通过一个例子来演示下它们的用法：

```
ZEND_FUNCTION(sample_array)
{
    zval *subarray;

    array_init(return_value);

    /* Add some scalars */
    add_assoc_long(return_value, "life", 42);
    add_index_bool(return_value, 123, 1);
    add_next_index_double(return_value, 3.1415926535);

    /* Toss in a static string, dup'd by PHP */
    add_next_index_string(return_value, "Foo", 1);

    /* Now a manually dup'd string */
    add_next_index_string(return_value, estrdup("Bar"), 0);

    /* Create a subarray */
    MAKE_STD_ZVAL(subarray);
    array_init(subarray);

    /* Populate it with some numbers */
    add_next_index_long(subarray, 1);
    add_next_index_long(subarray, 20);
    add_next_index_long(subarray, 300);

    /* Place the subarray in the parent */
    add_index_zval(return_value, 444, subarray);
}
```

这时如果我们用户端var_dump这个函数的返回值便会得到：

```
<?php
```

```
var_dump(sample_array());
```

输出：

```
array(6)
{
    ["life"]=> int(42)
    [123]=> bool(true)
    [124]=> float(3.1415926535)
    [125]=> string(3) "Foo"
    [126]=> string(3) "Bar"
    [444]=> array(3)
    {
        [0]=> int(1)
        [1]=> int(20)
        [2]=> int(300)
    }
}
```

links

- 8.2 操作HashTable的API
- 8.4 第8章小结

8.4 使用HashTable与{数组}

我们用了很长的篇幅在这一章描述内核中的HashTable结构以及PHP中的数组实现。在接下来的时间中，我们会在它的基础上学习一下内核是怎样实现与管理PHP语言中的资源与类的。

links

- 8.3 在内核中操作PHP语言中数组
- 9 PHP中的资源类型

9 PHP中的资源类型

截止到现在，我们已经熟悉了PHP语言中的字符串、数字、布尔以及数组的数据类型了，接下来，我们将接触另外一种PHP独特的数据类型——资源（Resource）。

links

- 8.4 第8章小结
- 9.1 复合类型的数据——{资源}

9.1 PHP中的资源类型

讲述之前，先描述下{资源}类型在内核中的结构：
````c //每一个资源都是通过它来实现的。 typedef struct _zend_rsrc_list_entry { void *ptr; int type; int refcount; }zend_rsrc_list_entry; ```` 在真实世界中，我们经常需要操作一些不好用标量值表现的数据，比如某个文件的句柄，而对于C来说，它也仅仅是个指针而已。 ````c #include int main(void) { FILE *fd; fd = fopen("/home/jdoe/.plan", "r"); fclose(fd); return 0; } ```` C语言中stdio的文件描述符(file descriptor)是与每个打开的文件相匹配的一个变量，它实际上是一个FILE类型的指针，它将在程序与硬件交互通讯时使用。我们可以使用fopen函数来打开一个文件获取句柄，之后只需把这个句柄传递给feof()、fread()、fwrite()、fclose()之类的函数，便可以对这个文件进行后续操作了。既然这个数据在C语言中就无法直接用标量数据来表示，那我们如何对其进行封装才能保证用户在PHP语言中也能使用到它呢？这便是PHP中资源类型变量的作用了！所以它也是通过一个zval结构来进行封装的。 资源类型的实现并不复杂，它的值其实仅仅是一个整数，内核将根据这个整数值去一个类似资源池的地方寻找最终需要的数据。 ### 资源类型变量的使用 资源类型的变量在实现中也是有类型区分的！为了区分不同类型的资源，比如一个是文件句柄，一个是mysql链接，我们需要为其赋予不同的分类名称。首先，我们需要先把这个分类添加到程序中去。这一步的操作可以在MINIT中来做： ````c #define PHP_SAMPLE_DESCRIPTOR_RES_NAME "山寨文件描述符" static int le_sample_descriptor; ZEND_MINIT_FUNCTION(sample) { le_sample_descriptor = zend_register_list_destructors_ex(NULL, NULL, PHP_SAMPLE_DESCRIPTOR_RES_NAME,module_number); return SUCCESS; } //附加资料 #define register_list_destructors(ld, pld) zend_register_list_destructors((void (*)(void *))ld, (void (*)(void *))pld, module_number); ZEND_API int zend_register_list_destructors(void (*ld)(void *), void (*pld)(void *), int module_number); ZEND_API int zend_register_list_destructors_ex(rsrc_dtor_func_t ld, rsrc_dtor_func_t pld, char *type_name, int module_number); ```` 接下来，我们把定义好的MINIT阶段的函数添加到扩展的module_entry里去，只需要把原来的"NULL, /* MINIT */"一行替换掉即可： ````c ZEND_MINIT(sample), /* MINIT */ ```` ZEND_MINIT_FUNCTION()宏用来帮助我们定义MINIT阶段的函数，这我们已经在第一章里描述过了，但将会在第12章和第三章有更详细的描述。 What's important to know at this juncture is that the MINIT method is executed once when your extension is first loaded and before any requests have been received. Here you've used that opportunity to register destructor functionsthe NULL values, which you'll change soon enoughfor a resource type that will be thereafter known by a unique integer ID. 看到zend_register_list_destructors_ex()函数，你肯定会想是不是也存在一个zend_register_list_destructors()函数呢？是的，确实有这么一个函数，它的参数中比前者少了资源类别的名称。那这两这的区别在哪呢？ ````php eaco $re_1; //resource(4) of type (山寨版File句柄) echo $re_2; //resource(4) of type (Unknown) ```` ### 创建资源 我们在上面向内核中注册了一种新的资源类型，下一步便可以创建这种类型的资源变量了。接下来让我们简单的重新实现一个fopen函数，现在叫sample_open： ````c PHP_FUNCTION(sample_fopen) { FILE *fp; char *filename, *mode; int filename_len, mode_len; if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",&filename, &filename_len,&mode, &mode_len) == FAILURE) { RETURN_NULL(); } if (!filename_len || !mode_len) { php_error_docref(NULL TSRMLS_CC, E_WARNING,"Invalid filename or mode length"); RETURN_FALSE; } fp = fopen(filename, mode); if (!fp) { php_error_docref(NULL TSRMLS_CC, E_WARNING,"Unable to open %s using mode %s",filename, mode); RETURN_FALSE; } //将fp添加到资源池中去，并标记它为le_sample_descriptor类型的。 ZEND_REGISTER_RESOURCE(return_value,fp,le_sample_descriptor); } ```` 如果前面章节的知识你都看过的话，应该可以猜出最后一行代码是干啥的了。它创建了一个新的le_sample_descriptor类型的资源，此资源的值是fp，另外它把这个资源加入到一个存储资源的HashTable中，并把此资源在其中对应的数字Key赋给return_value。
````

资源并不局限于文件句柄，我们可以申请一块内存，并且指向它的指针来作为一种资源。所以资源可以对应任意类型的数据。
销毁资源 世间万物皆有喜有悲，有生有灭，到了我们探讨如何销毁资源的时候了。最简单的一种莫过于仿照fclose写一个sample_close()函数，在它里面实现对某种{资源：专指PHP的资源类型变量代表的值}的释放。 但是，如果用户端的脚本通过unset()函数来释放某个资源类型的变量会如何呢？它们可不知道它的值最终对应一个FILE*指针啊，所以也无法使用fclose()函数来释放它，这个FILE*句柄很有可能会一直存在于内存中，直到PHP程序挂掉，由OS来回收。但在一个平常的Web环境中，我们的服务器都会长时间运行的。 难道就没有解决方案了吗？当然不是，谜底就在那个NULL参数里，就是我们在上面为了生成新的资源类型，调用的zend_register_list_destructors_ex()函数的第一个参数和第二个参数。这两个参数都各自代表一个回调参数。第一个回调函数会在脚本中的相应类型的资源变量被释放掉的时候触发，比如作用域结束了，或者被unset()掉了。 第二个回调函数则是用在一个类似于长链接类型的资源上的，也就是这个资源创建后会一直存在于内存中，而不会在request结束后被释放掉。它将会在Web服务器进程终止时调用，相当于在MSHUTDOWN阶段被内核调用。有关persistent resources的事宜，我们将在下一节里详述。 我们先来定义第一种回调函数。 ````c static void php_sample_descriptor_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC) { FILE *fp = (FILE*)rsrc->ptr; fclose(fp); } ```` 然后用它替换掉zend_register_list_destructors_ex()函数的第一个参数NULL： ````c le_sample_descriptor = zend_register_list_destructors_ex(php_sample_descriptor_dtor, NULL, PHP_SAMPLE_DESCRIPTOR_RES_NAME, module_number); ```` 现在，如果脚本中得到了一个上述类型的资源变量，当它被unset的时候，或者因为作用域执行完被内核释放掉的时候都会被内核调用底层的php_sample_descriptor_dtor来预处理它。这样一来，貌似我们根本就不需要sample_close()函数了！ ````php ```` unset($fp)执行后，内核会自动的调用php_sample_descriptor_dtor函数来清理这个变量对应的一些数据。 当然，事情绝对没有这么简单，让我们先记住这个疑问，继续往下看。 ### Decoding Resources 我们把资源变量比作书签，可如果仅有书签的话绝对没有任何作用啊！我们需要通过书签找到相应的页才行。对于资源变量，我们必须能够通过它找到相应的最终数据才行！ ````c ZEND_FUNCTION(sample_fwrite) { FILE *fp; zval *file_resource; char *data; int data_len; if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",&file_resource, &data, &data_len) == FAILURE) { RETURN_NULL(); } /* Use the zval* to verify the resource type and * retrieve its pointer from the lookup table */ ZEND_FETCH_RESOURCE(fp,FILE*,&file_resource,-1,PHP_SAMPLE_DESCRIPTOR_RES_NAME,le_sample_descriptor); /* Write the data, and * return the number of bytes which were * successfully written to the file */ RETURN_LONG(fwrite(data, 1, data_len, fp)); } ```` zend_parse_parameters()函数中的r占位符代表着接收资源类型的变量，它的载体是一个zval*。然后让我们看一下ZEND_FETCH_RESOURCE()宏函数。 ````c #define ZEND_FETCH_RESOURCE(rsrc, rsrc_type, passed_id,default_id, resource_type_name, resource_type) rsrc = (rsrc_type) zend_fetch_resource(passed_id TSRMLS_CC,default_id, resource_type_name, NULL,1, resource_type); ZEND_VERIFY_RESOURCE(rsrc); //在我们的例子中，它是这样的： fp = (FILE*) zend_fetch_resource(&file_descriptor TSRMLS_CC, -1,PHP_SAMPLE_DESCRIPTOR_RES_NAME, NULL,1, le_sample_descriptor); if (!fp) { RETURN_FALSE; } ```` zend_fetch_resource()是对zend_hash_find()的一层封装，它使用一个数字key去一个保存各种{资源}的HashTable中寻找最终需要的数据，找到之后，我们用ZEND_VERIFY_RESOURCE()宏函数校验一下这个数据。从上面的代码中我们可以看出，NULL、0是绝对不能作为一种资源的。 上面的例子中，zend_fetch_resource()函数首先获取le_sample_descriptor代表的资源类型，如果资源不存在或者接收的zval不是一个资源类型的变量，它便会返回NULL，并抛出相应的错误信息。 最后的ZEND_VERIFY_RESOURCE()宏函数如果检测到错误，便会自动返回，使我们可以从错误检测中脱离出来，更加专注于程序的主逻辑。现在我们已经获取到了相应的FILE*了，下面就用fwrite()向其中写入点数据吧！。

To avoid having zend_fetch_resource() generate an error on failure, simply pass NULL for the resource_type_name parameter. Without a meaningful error message to display, zend_fetch_resource() will fail silently instead.

我们也可以通过另一种方法来获取我们最终想要的数据。

```
ZEND_FUNCTION(sample_fwrite)
{
    FILE *fp;
    zval *file_resource;
    char *data;
    int data_len, rsrc_type;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",&file_resource, &data, &dat
        RETURN_NULL();
    }
    fp = (FILE*)zend_list_find(Z_RESVAL_P(file_resource),&rsrc_type);
    if (!fp || rsrc_type != le_sample_descriptor) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Invalid resource provided");
        RETURN_FALSE;
    }
    RETURN_LONG(fwrite(data, 1, data_len, fp));
}
```

可以根据自己习惯来选择到底使用哪一种形式，不过推荐使用ZEND_FETCH_RESOURCE()宏函数。

Forcing Destruction

在上面我们还有个疑问没有解决，就是类似于我们上面实现的unset($fp)真的是万能的么？当然不是，看一下下面的代码：

```
<?php
  $fp = sample_fopen("/home/jdoe/world_domination.log", "a");
  $evil_log = $fp;
  unset($fp);
?>
```

这次，$fp和$evil_log共用一个zval，虽然$fp被释放了，但是它的zval并不会被释放，因为$evil_log还在用着。也就是说，现在$evil_log代表的文件句柄仍然是可以写入的！所以为了避免这种错误，真的需要我们手动来close it！sample_close()函数是必须存在的！

```
PHP_FUNCTION(sample_fclose)
{
    FILE *fp;
    zval *file_resource;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r",&file_resource) == FAILURE )
        RETURN_NULL();
    }

    /* While it's not necessary to actually fetch the
     * FILE* resource, performing the fetch provides
```

```
     * an opportunity to verify that we are closing
     * the correct resource type. */
    ZEND_FETCH_RESOURCE(fp, FILE*, &file_resource, -1,PHP_SAMPLE_DESCRIPTOR_RES_NAME, le_

    /* Force the resource into self-destruct mode */
    zend_hash_index_del(&EG(regular_list),Z_RESVAL_P(file_resource));
    RETURN_TRUE;
}
```

这个删除操作也再次说明了资源数据是保存在HashTable中的。虽然我们可以通过zend_hash_index_find()或者zend_hash_next_index_insert()之类的函数操作这个储存资源的HashTable，但这绝不是一个好主意，因为在后续的版本中，PHP可能会修改有关这一部分的实现方式，到那时上述方法便不起作用了，所以为了更好的兼容性，请使用标准的宏函数或者api函数。 当我们在EG(regular_list)这个HashTable中删除数据的时候，回调用一个dtor函数，它根据资源变量的类别来调用相应的dtor函数实现，就是我们调用zend_register_list_destructors_ex()函数时的第一个参数。

在很多地方，我们都会看到一个专门用来删除的zend_list_delete()宏函数，因为它考虑了资源数据自己的引用计数，所以我们将在后面的章节中介绍它。

links

- 9 PHP中的资源类型
- 9.2 Persistent Resources

9.2 PHP中的资源类型

通常情况下，像{资源}这类复合类型的数据都会占用大量的硬件资源，比如内存、CPU以及网络带宽。对于使用频率超级高的数据库链接，我们可以获取一个长链接，使其不会在脚本结束后自动销毁，一旦创建便可以在各个请求中直接使用，从而减少每次创建它的消耗。Mysql的长链接在PHP内核中其实就是一种持久{资源}。 Memory Allocation 前面的章节里我们接触了emalloc()之类的以e开头的内存管理函数，通过它们申请的内存都会被内核自动的进行垃圾回收的操作。而对于一个持久{资源}来说，我们是绝对不希望它在脚本结束后被回收的。

假设我们需要在我们的{资源}中同时保存文件名和文件句柄两个数据，现在我们就需要自己定义个结构了：

```
typedef struct _php_sample_descriptor_data
{
    char *filename;
    FILE *fp;
}php_sample_descriptor_data;
```

当然，因为结构变了(之前是个FILE*)，我们之前的代码也需要跟着改动。这里还没有涉及到持久{资源}，仅仅是换了一种{资源}结构

```
static void php_sample_descriptor_dtor(zend_rsrc_list_entry *rsrc TSRMLS_DC)
{
    php_sample_descriptor_data *fdata = (php_sample_descriptor_data*)rsrc->ptr;
    fclose(fdata->fp);
    efree(fdata->filename);
    efree(fdata);
}
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode;
    int filename_len, mode_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",&filename, &filename_len,&m
    {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Invalid filename or mode length");
        RETURN_FALSE;
    }
    fp = fopen(filename, mode);
    if (!fp)
    {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Unable to open %s using mode %s",file
        RETURN_FALSE;
    }
    fdata = emalloc(sizeof(php_sample_descriptor_data));
    fdata->fp = fp;
    fdata->filename = estrndup(filename, filename_len);
    ZEND_REGISTER_RESOURCE(return_value, fdata,le_sample_descriptor);
```

```
}
PHP_FUNCTION(sample_fwrite)
{
    php_sample_descriptor_data *fdata;
    zval *file_resource;
    char *data;
    int data_len;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "rs",&file_resource, &data, &dat
    {
        RETURN_NULL();
    }
    ZEND_FETCH_RESOURCE(fdata, php_sample_descriptor_data*,&file_resource, -1,PHP_SAMPLE_
    RETURN_LONG(fwrite(data, 1, data_len, fdata->fp));
}
```

我们这里没有重写sample_fclose()函数，你可以尝试着自己实现它。
现在编译运行，所有代码的结果都非常正确，我们还可以在内核中获取每个{资源}对应的文件名称了。

```
PHP_FUNCTION(sample_fname)
{
    php_sample_descriptor_data *fdata;
    zval *file_resource;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "r",&file_resource) == FAILURE )
    {
        RETURN_NULL();
    }
    ZEND_FETCH_RESOURCE(fdata, php_sample_descriptor_data*,&file_resource, -1,PHP_SAMPLE_
    RETURN_STRING(fdata->filename, 1);
}
```

现在，Persistent Resources来了！

Delayed Destruction

在前面我们删除一个{资源}的时候，其实是去EG(regular_list)中将其删掉，EG(regular_list)存储着所有的只用在当前请求的{资源}。

持久{资源},存储在另一个HashTable中：EG(persistent_list)。其与EG(regular_list)有个明显的区别，那就是它每个值的索引都是字符串类型的，而且它的每个值也不会在每次请求结束后被释放掉，只能我们手动通过zend_hash_del()来删除，或者在进程结束后类似于MSHUTDOWN阶段将EG(persistent_list)整体清除，最常见的情景便是操作系统关闭了Web Server。 EG(persistent_list)对其元素也有自己的dtor回调函数，和EG(regular_list)一样，它将根据其值的类型去调用不同的回调函数，我们这一次注册回调函数的时候，需要用到zend_register_list_destructors_ex()函数的第二个参数，第一个则被赋成NULL。 在底层的实现中，持久的和regular{资源}是分别在不同的地方存储的，也分别拥有各自不同的释放函数。但在我们为脚本提供的函数中，却希望能够封装这种差异，从而使我们的用户使用起来更加方便快捷。

```
static int le_sample_descriptor_persist;

static void php_sample_descriptor_dtor_persistent(zend_rsrc_list_entry *rsrc TSRMLS_DC)
```

```
{
    php_sample_descriptor_data *fdata = (php_sample_descriptor_data*)rsrc->ptr;
    fclose(fdata->fp);
    pefree(fdata->filename, 1);
    pefree(fdata, 1);
}

PHP_MINIT_FUNCTION(sample)
{
    le_sample_descriptor = zend_register_list_destructors_ex(php_sample_descriptor_dtor,
    le_sample_descriptor_persist =zend_register_list_destructors_ex(NULL, php_sample_desc
    return SUCCESS;
}
```

我们并没有为这两种{资源}起不同的名字，以防使用户产生疑惑。 现在我们的PHP扩展中引进了一种新的{资源}，所以我们需要改写一下上面的函数，尽量使用户使用时感觉不到这种差异。

```
//sample_fopen()
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode;
    int filename_len, mode_len;
    zend_bool persist = 0;

    //类比一下mysql_connect函数的最后一个参数。
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,"ss|b",&filename, &filename_len,
    {
        RETURN_NULL();
    }
    if (!filename_len || !mode_len)
    {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Invalid filename or mode length");
        RETURN_FALSE;
    }

    fp = fopen(filename, mode);
    if (!fp)
    {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Unable to open %s using mode %s",file
        RETURN_FALSE;
    }

    if (!persist)
    {
        fdata = emalloc(sizeof(php_sample_descriptor_data));
        fdata->filename = estrndup(filename, filename_len);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,le_sample_descriptor);
    }
    else
    {
        list_entry le;
        char *hash_key;
        int hash_key_len;
```

```
        fdata =pemalloc(sizeof(php_sample_descriptor_data),1);
        fdata->filename = pemalloc(filename_len + 1, 1);
        memcpy(data->filename, filename, filename_len + 1);
        fdata->fp = fp;

        //在EG(regular_list中存一份)
        ZEND_REGISTER_RESOURCE(return_value, fdata,le_sample_descriptor_persist);

        //在EG(persistent_list)中再存一份
        le.type = le_sample_descriptor_persist;
        le.ptr = fdata;
        hash_key_len = spprintf(&hash_key, 0,"sample_descriptor:%s:%s", filename, mode);
        zend_hash_update(&EG(persistent_list),hash_key, hash_key_len + 1,(void*)&le, size
        efree(hash_key);
    }
}
```

在持久{资源}时，因为我们在EG(regular_list)中也保存了一份，所以脚本中我们资源类型的变量在实现中仍然是保存着一个resource ID，我们可以用它来进行之前章节所做的工作。 将其添加到EG(persistent_list)中时，我们进行的操作流程几乎和ZEND_REGISTER_RESOURCE()宏函数一样，唯一的不同便是索引由之前的数字类型换成了字符串类型。 当一个保存在EG(regular_list)中的持久{资源}被脚本释放时，内核会在EG(regular_list)寻找它对应的dtor函数，但它找到的是NULL，因为我们在使用zend_register_list_destructors_ex()函数声明这种资源类型时，第一个参数的值为NULL。所以此时这个{资源}不会被任何dtor函数调用，可以继续存在于内存中，任脚本流逝，请求更迭。 当web server的进程执行完毕后，内核会扫描EG(persistent_list)的dtor，并调用我们已经定义好的释放函数。在我们定义的释放函数中，一定要记得使用pfree函数来释放内存，而不是efree。

Reuse

创建持久{资源}的目的是为了使用它，而不是让它来浪费内存的，我们再次重写一下sample_open()函数，这一次我们将检测需要创建的资源是否已经在persistent_list中存在了。

```
PHP_FUNCTION(sample_fopen)
{
    php_sample_descriptor_data *fdata;
    FILE *fp;
    char *filename, *mode, *hash_key;
    int filename_len, mode_len, hash_key_len;
    zend_bool persist = 0;
    list_entry *existing_file;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,"ss|b",&filename, &filename_len,
    {
        RETURN_NULL();
    }

    if (!filename_len || !mode_len)
    {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Invalid filename or mode length");
        RETURN_FALSE;
    }

    //看看是否已经存在，如果已经存在就直接使用，不再创建
```

```
    hash_key_len = spprintf(&hash_key, 0,"sample_descriptor:%s:%s", filename, mode);
    if (zend_hash_find(&EG(persistent_list), hash_key,hash_key_len + 1, (void **)&existin
    {
        //存在一个，直接使用！
        ZEND_REGISTER_RESOURCE(return_value,existing_file->ptr, le_sample_descriptor_pers
        efree(hash_key);
        return;
    }

    fp = fopen(filename, mode);
    if (!fp)
    {
        php_error_docref(NULL TSRMLS_CC, E_WARNING,"Unable to open %s using mode %s",file
        RETURN_FALSE;
    }
    if (!persist)
    {
        fdata = emalloc(sizeof(php_sample_descriptor_data));
        fdata->filename = estrndup(filename, filename_len);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,le_sample_descriptor);
    }
    else
    {
        list_entry le;
        fdata =pemalloc(sizeof(php_sample_descriptor_data),1);
        fdata->filename = pemalloc(filename_len + 1, 1);
        memcpy(data->filename, filename, filename_len + 1);
        fdata->fp = fp;
        ZEND_REGISTER_RESOURCE(return_value, fdata,le_sample_descriptor_persist);

        /* Store a copy in the persistent_list */
        le.type = le_sample_descriptor_persist;
        le.ptr = fdata;

        //hash_key在上面已经被创建了
        zend_hash_update(&EG(persistent_list),hash_key, hash_key_len + 1,(void*)&le, size
    }
    efree(hash_key);
}
```

因为所有的PHP扩展都共用同一个HashTable来保存持久{资源}，所以我们在为{资源}的索引起名时，一定要唯一，同时必须简单，方便我们在其它的函数中构造出来。

Liveness Checking and Early Departure

一旦我们打开一个本地文件，便可以一直占有它的操作句柄，保证随时可以打开它。但是对于一些存在于远程计算机上的资源，比如mysql链接、http链接，虽然我们仍然握着与服务器的链接，但是这个链接在服务器端可能已经被关闭了，在本地我们就无法再用它来做一些有价值的工作了。

所以，当我们使用{资源}，尤其是持久{资源}时，一定要保证获取出来的{资源}仍然是有效的、可以使用的。如果它失效了，我们必须将其从persistent list中移除。下面就是一个检测socket有效性的例子：

```
if (zend_hash_find(&EG(persistent_list), hash_key,hash_key_len + 1, (void**)&socket) == S
```

```
{
    if (php_sample_socket_is_alive(socket->ptr))
    {
        ZEND_REGISTER_RESOURCE(return_value,socket->ptr, le_sample_socket);
        return;
    }
    zend_hash_del(&EG(persistent_list),hash_key, hash_key_len + 1);
}
```

如你所见，{资源}失效后，我们只要把它从HashTable中删除就行了，这一步操作同样会激活我们设置的回调函数。On completion of this code block, the function will be in the same state it would have been if no resource had been found in the persistent list.

Agnostic Retrieval

现在我们已经可以创建资源类型并生成新的资源，还能将持久{资源}与平常{资源}使用的差异性封装起来。但是如果用户对一个持久{资源}调用sample_fwrite()时候并不会正常工作，先想一下内核是如何通过一个数字所以在regular_list中获取最终资源的。

```
ZEND_FETCH_RESOURCE(
    fdata,
    php_sample_descriptor_data*,
    &file_resource,
    -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME,
    le_sample_descriptor
);
```

le_sample_descriptor可以保证你获取到的资源确实是这种类型的，绝不会出现你想要一个文件句柄，却返回给你一个mysql链接的情况。这种验证是必须的，但有时你又想绕过这种验证，因为我们放在persistenst_list中的{资源}是le_sample_descruotor_persist类型的，所以当我们把它复制到regular_list中时，它也是le_sample_descructor_persist的，所以如果我们想获取它，貌似只有两种方法，要么修改类型，要么再写一个新的sample_write_persistent函数的实现。或者极端一些，在sample_write函数里进行复杂的判断。但是如果sample_write()函数能同时接收它们两种类型的{资源}多好啊....

事情没有这么复杂，我们确实可以在sample_write()函数里获取{资源}时候同时指定两种类型。那就是使用ZEND_FETCH_RESOURCE2()宏函数，它与ZEND_FETCH_RESOURCE()宏函数的唯一区别就是它可以接收两种类型参数。

```
ZEND_FETCH_RESOURCE2(
    fdata,
    php_sample_descriptor_data*,
    &file_resource,
    -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME,
    le_sample_descriptor,
    le_sample_descriptor_persist
);
```

现在，只要resource ID对应的最终资源类型是persistent或者non-persistent的一种便可以正常通过验证了。

什么，你想设置三种甚至更多的类型？!!那你只能直接使用zend_fetch_resource()函数了。

```
//一种类型的
fp = (FILE*) zend_fetch_resource(
        &file_descriptor TSRMLS_CC,
        -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME,
        NULL,
        1,
        le_sample_descriptor
);
ZEND_VERIFY_RESOURCE(fp);
```

想看看ZEND_FETCH_RESOURCE2()宏函数的实现么？

```
//两种类型的
fp = (FILE*) zend_fetch_resource(
        &file_descriptor TSRMLS_CC,
        -1,
        PHP_SAMPLE_DESCRIPTOR_RES_NAME,
        NULL,
        2,
        le_sample_descriptor,
        le_sample_descriptor_persist
);
ZEND_VERIFY_RESOURCE(fp);
```

再给力一些，三种类型的：

```
fp = (FILE*) zend_fetch_resource(
    &file_descriptor TSRMLS_CC,
    -1,
    PHP_SAMPLE_DESCRIPTOR_RES_NAME,
    NULL,
    3,
    le_sample_descriptor,
    le_sample_descriptor_persist,
    le_sample_othertype
);
ZEND_VERIFY_RESOURCE(fp);
```

话都说到这份上了，你肯定知道四种、五种、更多种类型的应该怎么调用了。

links

- 9.1 复合类型的数据——{资源}
- 9.3 {资源}自有的引用计数

9.3 PHP中的资源类型

zval通过引用计数来节省内存的，这个我们都知道了，但你可能不知道的是，某个zval对应的{资源}在实现时也使用了引用计数这种概念，也就是有了两种引用计数！

{资源}对应的zval的类型是IS_RESOURCE，它并不保存最终的数据，而只保存一个数字，即EG(regular_list)中的数字索引。

当{资源}被创建时，比如我们调用sample_fopen()函数：

```
$a = sample_fopen('notes.txt', 'r');
//此时：var->refcount__gc = 1, rsrc->refcount = 1

$b = $a;
//此时：var->refcount__gc = 2, rsrc->refcount = 1

unset($b);
//此时：var->refcount__gc = 1, rsrc->refcount = 1

/*
 下面来个复杂的！
 */

$b = $a;
$c = &$a;
//此时：
/*
    bvar->refcount = 1, bvar->is_ref = 0
    acvar->refcount = 2, acvar->is_ref = 1
    rsrc->refcount = 2
*/
```

现在，如果我们unset($b)，内核只会把rsrc->refcount的值减1。只有当rsrc->refcount的值为0时，我们预设的dtor释放函数才会被激活并调用。

links

- 9.2 Persistent Resources
- 9.4 第九章小结

9.4 PHP中的资源类型

通过这一章介绍的技术，我们已经可以使用PHP中{资源}了，这将使我们更容易的在扩展中使用一些第三方库，比如使用zip扩展时，我们需要把它的一些特殊的量封装成资源供脚本使用。这无疑极大的增强了PHP的威力！{资源}V5!

下一章将会说一下PHP中的对象！原书中分别讲述了PHP4与PHP5的实现，这里我没有参照原书的安排，按照自己的理解完全重写了这一部分，以PHP5为基础讲述，确切的说，是以PHP5.3.6讲述，这个版本问题我也已经在本书开头说明了，因为我私自认为PHP4已经不再重要了。

links

- 9.3 {资源}自有的引用计数
- 10 PHP中的面向对象（一）

10 PHP中的面向对象（一）

面向对象的概念这里就不再叙述了。原书中把这一部分的知识分开到PHP4和PHP5中来讲的，这里我做了大幅的调整，几乎是进行了重写。前一部分主要介绍了如何定义类、接口等一些声明类的操作。后一部分主要介绍了对象的使用等一些对实例的操作。

links

- 9.4 第九章小结
- 10.1 zend_class_entry

10.1 PHP中的面向对象（一）

zend_class_entry是内核中定义的一个结构体，是内核实现PHP语言中类与对象的一个非常基础、关键的结构类型。他就相当于我们定义的类的原型。 如果我们想获得一个名字为myclass的类该怎么做呢？首先我们定义一个zend_class_entry变量，并为它设置名字，最后注册到runtime中去。

```
zend_class_entry *myclass_ce;


static zend_function_entry myclass_method[] = {
    { NULL, NULL, NULL }
};

ZEND_MINIT_FUNCTION(sample3)
{
    zend_class_entry ce;

    //"myclass"是这个类的名称。
    INIT_CLASS_ENTRY(ce, "myclass",myclass_method);
    myclass_ce = zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

这样我们便定义了一个类myclass，而且我们可以正常的在PHP语言中使用它，比如：

```
<?php
$obj = new myclass();
```

我们上面还定义了一个myclass_ce指针，他是干什么用的呢？当我们在扩展中对这个类进行操作，比如生成实例的时候，会使用到它，它的作用就类似于打开文件的操作句柄。

links

- 10 PHP中的面向对象（一）
- 10.2 定义一个类

10.2 PHP中的面向对象（一）

在这一节中，我们正式的定义一个类。首先我给出PHP语言的实现：

```
<?php
class myclass
{
    public $public_var;
    private $private_var;
    protected $protected_var;

    public static $static_var;

    public function __construct()
    {
        echo "我是__construct方法\n";
    }

    public function public_method()
    {
        echo "我是public类型的方法\n";
    }

    public function private_method()
    {
        echo "我是private类型的方法\n";
    }

    public function protected_method()
    {
        echo "我是protected类型的方法\n";
    }

    public static function static_var()
    {
        echo "我是static类型的方法\n";
    }
}
```

定义类对应的zend_class_entry

定义类的第一步，便是先定义好这个类的zend_class_entry，这一步操作是在MINIT阶段完成的。

```
static zend_function_entry myclass_method[]=
{NULL,NULL,NULL};

PHP_MINIT_FUNCTION(test)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce,"myclass",myclass_method);
    zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

```
//这就是最简单的一个类，没有属性没有方法，但是可以使用了
/*
<?php
$obj = new myclass();
var_dump($obj);

//得到：object(myclass)#1 (0) {}

*/
```

某个类的zend_class_entry会经常用到，所以我们一般会把它保存在一个变量里，供扩展中其它地方的程序使用，所以上述的代码组合一般是这样的：

```
zend_class_entry *myclass_ce;

static zend_function_entry myclass_method[]={
    {NULL,NULL,NULL}
};

ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce,"myclass",myclass_method);
    myclass_ce = zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

为类定义属性

我们可以用zend_declare_property*系列函数来完成这项操作，为某个类定义属性一般会需要三个信息：

- 属性的名称
- 属性的默认值
- 属性的访问权限等

我们为上面的myclass类定义一个名为“public_var”的属性，默认值为null，访问权限为public。

```
ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce,"myclass",myclass_method);

    myclass_ce = zend_register_internal_class(&ce TSRMLS_CC);

    //定义属性
    zend_declare_property_null(myclass_ce, "public_var", strlen("public_var"), ZEND_ACC_P
    return SUCCESS;
}
```

ZEND_ACC_PUBLIC是ZEND_ACC系列掩码中的一个，代表着public，其余的还有

ZEND_ACC_PRIVATE,ZEND_ACC_PROTECTED等等，详细描述请见后面的章节。这三个掩码比较简单，就不再叙述了。

为类定义方法

为类定义方法比较繁琐一些，首先我们先回顾一下zend_function_entry结构，在以前我们用它来保存我们扩展的函数，通过它把PHP语言中的函数和我们用C语言编写的函数联系起来，在这它也发挥了这么一个桥梁的作用。下面我们实现myclass类的public_method()和构造方法。

```
//首先，定义这个函数的C语言部分，不过这一次我们使用的是ZEND_METHOD
ZEND_METHOD( myclass , public_method )
{
    php_printf("我是public类型的方法\n");
}

ZEND_METHOD( myclass , __construct )
{
    php_printf("我是__construct方法\n");
}

//然后，用PHP_METHOD声明public_method和__construct。

PHP_METHOD(myclass, public_method);
PHP_METHOD(myclass, __construct);

//再定义一个zend_function_entry
zend_function_entry myclass_method[]=
{
    ZEND_ME(myclass,    public_method,    NULL,    ZEND_ACC_PUBLIC)
    ZEND_ME(myclass,    __construct,    NULL,    ZEND_ACC_PUBLIC|ZEND_ACC_CTOR)
    {NULL,    NULL,    NULL}
}

//最后，在MINIT阶段register internal class的时候将它作为一个参数传递进去
ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce;

    //这里使用了myclass_method这个zend_function_entry
    INIT_CLASS_ENTRY(ce,"myclass",myclass_method);

    myclass_ce = zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

现在我们在PHP脚本中调用一下这个方法，看看输出结果：

```
<?php
$obj = new myclass();
$obj->public_method();

/*
walu@walu-ThinkPad-Edge:/cnan/program/php-5.3.6/ext/test$ php test.php
我是__construct方法
```

```
我是public_method方法
*/
```

这里在定义__construct方法的时候，使用到了ZEND_ACC_CTOR，它的作用便是声明这个方法是此类的构造函数，而ZEND_ACC_PUBLIC|ZEND_ACC_CTOR是我们常见的掩码或运算，代表它是一个public类型构造函数，:-)。如果我们去掉ZEND_ACC_CTOR标志，那么此构造函数还会起作用吗？在这里的例子中它仍然起作用，但是在别的环境下我就不敢保证了。 说到现在，protected和private类型的属性与方法的定义和public的一样。而定义static的属性与方法只是在掩码标志中加入ZEND_ACC_STATIC即可。 下面详细的罗列出了所有掩码，fn_flags代表可以在定义方法时使用，zend_property_info.flags代表可以在定义属性时使用，ce_flags代表在定义zend_class_entry时候可用。

```
#define ZEND_ACC_STATIC                     0x01     /* fn_flags, zend_property_info.flag
#define ZEND_ACC_ABSTRACT                   0x02     /* fn_flags */
#define ZEND_ACC_FINAL                      0x04     /* fn_flags */
#define ZEND_ACC_IMPLEMENTED_ABSTRACT       0x08     /* fn_flags */
#define ZEND_ACC_IMPLICIT_ABSTRACT_CLASS    0x10     /* ce_flags */
#define ZEND_ACC_EXPLICIT_ABSTRACT_CLASS    0x20     /* ce_flags */
#define ZEND_ACC_FINAL_CLASS                0x40     /* ce_flags */
#define ZEND_ACC_INTERFACE                  0x80     /* ce_flags */
#define ZEND_ACC_INTERACTIVE                0x10     /* fn_flags */
#define ZEND_ACC_PUBLIC                     0x100    /* fn_flags, zend_property_info.flag
#define ZEND_ACC_PROTECTED                  0x200    /* fn_flags, zend_property_info.flag
#define ZEND_ACC_PRIVATE                    0x400    /* fn_flags, zend_property_info.flag
#define ZEND_ACC_PPP_MASK    (ZEND_ACC_PUBLIC | ZEND_ACC_PROTECTED | ZEND_ACC_PRIVATE)
#define ZEND_ACC_CHANGED                    0x800    /* fn_flags, zend_property_info.flag
#define ZEND_ACC_IMPLICIT_PUBLIC            0x1000   /* zend_property_info.flags; unused
#define ZEND_ACC_CTOR                       0x2000   /* fn_flags */
#define ZEND_ACC_DTOR                       0x4000   /* fn_flags */
#define ZEND_ACC_CLONE                      0x8000   /* fn_flags */
#define ZEND_ACC_ALLOW_STATIC               0x10000  /* fn_flags */
#define ZEND_ACC_SHADOW                     0x20000  /* fn_flags */
#define ZEND_ACC_DEPRECATED                 0x40000  /* fn_flags */
#define ZEND_ACC_CLOSURE                    0x100000 /* fn_flags */
#define ZEND_ACC_CALL_VIA_HANDLER           0x200000 /* fn_flags */
```

ZEND_ACC_CTOR与ZEND_ACC_DTOR是比较特殊的掩码标志，分别代表着构造函数与析构函数，不要将这两个标志位用在其它的方法上面。其它的一些魔术方法，如**get**,call等大都需要arginfo，有关它们的内容将在下一章中描述。

为类定义常量

这个内容比较简单，只涉及到一组函数，可以查看Zend/zend_API.h

```
ZEND_API int zend_declare_class_constant(zend_class_entry *ce, const char *name, size_t n
ZEND_API int zend_declare_class_constant_null(zend_class_entry *ce, const char *name, siz
ZEND_API int zend_declare_class_constant_long(zend_class_entry *ce, const char *name, siz
ZEND_API int zend_declare_class_constant_bool(zend_class_entry *ce, const char *name, siz
ZEND_API int zend_declare_class_constant_double(zend_class_entry *ce, const char *name, s
ZEND_API int zend_declare_class_constant_stringl(zend_class_entry *ce, const char *name,
ZEND_API int zend_declare_class_constant_string(zend_class_entry *ce, const char *name, s
```



links

- 10.1 zend_class_entry
- 10.3 定义一个接口

10.3 PHP中的面向对象（一）

定义一个接口还是很方便的，我先给出一个PHP语言中的形式。

```
<?php
interface i_myinterface
{
    public function hello();
}
```

那它在扩展中的实现是这样的。

```
zend_class_entry *i_myinterface_ce;

static zend_function_entry i_myinterface_method[]={
    ZEND_ABSTRACT_ME(i_myinterface, hello, NULL) //注意这里的null指的是arginfo
    {NULL,NULL,NULL}
};

ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce, "i_myinterface", i_myinterface_method);

    i_myinterface_ce = zend_register_internal_interface(&ce TSRMLS_CC);
    return SUCCESS;
}
```

我们使用ZEND_ABSTRACT_ME()宏函数来为这个接口添加函数，它的作用是声明一个类似虚函数的东西，不用实现。也就是说我们不用为其添加ZEND_METHOD(i_myinterface,hello){...}的实现。但是这个宏函数只能为我们实现public类型函数的声明，如果有其它特殊需要，需要使用ZEND_FENTRY()宏函数来实现，因为ZEND_ABSTRACT_ME只不过是后者的一种封装。 下面我们在PHP语言中使用这个接口

```
<?php
class sample implements i_myinterface
{
    public $name = "hello world!";

    public function hello()
    {
        echo $this->name."\n";
    }
}

$obj = new sample();
$obj->hello();
```

links

- 10.2 定义一个类
- 10.4 继承与实现接口

10.4 PHP中的面向对象（一）

在定义一个类时往往会使其继承某个父类或者实现某个接口，在扩展中实现这个功能非常方便。下面我先给出PHP语言中的代码。

```
<?php
interface i_myinterface
{
    public function hello();
}

class parent_class implements i_myinterface
{
    public function hello()
    {
        echo "Good Morning!\n";
    }
}

final class myclass extends parent_class
{
    public function call_hello()
    {
        $this->hello();
    }
}
```

上面的代码我们已经非常熟悉了，它们在PHP扩展中的实现应该是这样的：

```

//三个zend_class_entry
zend_class_entry *i_myinterface_ce,*parent_class_ce,*myclass_ce;

//parent_class的hello方法
ZEND_METHOD(parent_class,hello)
{
    php_printf("hello world!\n");
}

//myclass的call_hello方法
ZEND_METHOD(myclass,call_hello)
{
    //这里涉及到如何调用对象的方法，详细内容下一章叙述
    zval *this_zval;
    this_zval = getThis();
    zend_call_method_with_0_params(&this_zval,myclass_ce,NULL,"hello",NULL);
}

//各自的zend_function_entry
static zend_function_entry i_myinterface_method[]={
    ZEND_ABSTRACT_ME(i_myinterface,    hello,    NULL)
    {NULL,NULL,NULL}
};
```

```
static zend_function_entry parent_class_method[]={
    ZEND_ME(parent_class,hello,NULL,ZEND_ACC_PUBLIC)
    {NULL,NULL,NULL}
};

static zend_function_entry myclass_method[]={
    ZEND_ME(myclass,call_hello,NULL,ZEND_ACC_PUBLIC)
    {NULL,NULL,NULL}
};

ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce,p_ce,i_ce;
    INIT_CLASS_ENTRY(i_ce,"i_myinterface",i_myinterface_method);
    i_myinterface_ce = zend_register_internal_interface(&i_ce TSRMLS_CC);


    //定义父类，最后使用zend_class_implements函数声明它实现的接口
    INIT_CLASS_ENTRY(p_ce,"parent_class",parent_class_method);
    parent_class_ce = zend_register_internal_class(&p_ce TSRMLS_CC);
    zend_class_implements(parent_class_ce TSRMLS_CC,1,i_myinterface_ce);

    //定义子类，使用zend_register_internal_class_ex函数
    INIT_CLASS_ENTRY(ce,"myclass",myclass_method);
    myclass_ce = zend_register_internal_class_ex(&ce,parent_class_ce,"parent_class" TSRML
    //注意：ZEND_ACC_FINAL是用来修饰方法的，而ZEND_ACC_FINAL_CLASS是用来修饰类的
    myclass_ce->ce_flags |= ZEND_ACC_FINAL_CLASS;
    return SUCCESS;
}
```

这样，当我们在PHP语言中进行如下操作时，便会得到预期的输出：

```
<?php
$obj = new myclass();
$obj->hello();
/*
输出内容：
walu@walu-ThinkPad-Edge:/cnan/program/php-5.3.6/ext/test$ php test.php
hello world!
*/
```

这里的ZEND_ABSTRACT_ME()宏函数比较特殊，它会声明一个abstract public类型的函数，这个函数不需要我们实现，因此也就不需要相应的ZEND_METHOD(i_myinterface,hello){...}的实现。一般来说，一个接口是不能设计出某个非public类型的方法的，因为接口暴露给使用者的都应该是一些公开的信息。不过如果你非要这么设计，那也不是办不到，只要别用ZEND_ABSTRACT_ME()宏函数就行了，而用它的底层实现ZEND_FN()宏函数

```
//它可以对应<?php ...public static function apply_request();...的接口方法声明。
static zend_function_entry i_myinterface[]=
{
    ZEND_FENTRY(apply_request, NULL, NULL, ZEND_ACC_STATIC|ZEND_ACC_ABSTRACT|ZEND_ACC_PUB
    {NULL,NULL,NULL}
```

```
};
```

这样，只要掩码中有ZEND_ACC_ABSTRACT，便代表是一个不需要具体实现的方法。ZEND_FENTRY其实是ZEND_ME和ZEND_FE的最终实现，现在我们把这一组宏罗列在这一次展开，供你参考使用。

```
#define ZEND_FENTRY(zend_name, name, arg_info, flags)    { #zend_name, name, arg_info, (z

#define ZEND_FN(name) zif_##name

#define ZEND_MN(name) zim_##name

#define ZEND_FE(name, arg_info)                        ZEND_FENTRY(name, ZEND_FN(name), a

#define ZEND_ME(classname, name, arg_info, flags)    ZEND_FENTRY(name, ZEND_MN(classname#
```

links

- 10.3 定义一个接口
- 10.5 小结

10.5 PHP中的面向对象（一）

这一章是我自己写的，如果有什么错误，还请大家指正。这章主要介绍了类与接口的定义，在下一章将看一下如何对类进行操作，比如调用方法、修改属性等。

links

- 10.4 继承与实现接口
- 11 PHP中的面向对象（二）

11 PHP中的面向对象（二）

上一章里，我们看了一下如何在PHP扩展里定义类与接口，那这一章里我们将入手学习一下如何在PHP扩展中操作类的实例————对象。 PHP语言中的面向对象其实是分为三个部分来实现的，class、object、refrence。class就是我们所说的类，可以直观的理解为前面章节中所描述的zend_class_entry。object就是实际的对象。每一个zval并不直接包含具体的object，而是通过一个索引--refrence与其联系。也就是说，每个class都有很多个object实例，并把他们统一的放在一个数组里，每个zval只要记住自己相应的key就行了。如此一来，我们在传递zval时候，实际上传递的是一个索引，而不是内存中具体的对象数据。

links

- 10.5 小结
- 11.1 生成对象的实例

11.1 PHP中的面向对象（二）

为了操作一个对象，我们需要先获取这个对象的实例，而这肯定会涉及调用对象的构造方法。有关如何在扩展中调用PHP的函数与对象的方法这里不展开描述了。 首先我们先了解下一个object在PHP内核中到底是如何实现的。

```
typedef struct _zend_object_value {
    zend_object_handle handle;
    zend_object_handlers *handlers;
} zend_object_value;

//此外再回顾一下zval的值value的结构。
typedef union _zvalue_value {
    long lval;                  /* long value */
    double dval;                /* double value */
    struct {
        char *val;
        int len;
    } str;
    HashTable *ht;              /* hash table value */
    zend_object_value obj;
} zvalue_value;
```

如果我们有一个zval *tmp，那么tmp->value.obj来访问到最终保存对象实例的zend_object_value结构体，它包含两个成员：

- zend_object_handle handle：最终实现是一个unsigned int值，Zend会把每个对象放进数组里，这个handle就是此实例的索引。所以我们在把对象当作参数传递时，只不过是传递的handle罢了，这样对性能有利，同时也是对象的引用机制的原理。
- zend_object_handlers *handlers：这个里面是一组函数指针，我们可以通过它来对对象进行一些操作，比如：添加引用、获取属性等。此结构体在Zend/zend_object_handlers.h里定义。

下面我给出这个类的PHP语言实现，让我们在扩展中实现它，并生成它。

```
<?php
class baby
{
    public function __construct()
    {
        echo "a new baby!\n";
    }

    public function hello()
    {
        echo "hello world!\n";
    }
}

function test_call()
{
    $obj = new baby();
```

```
    $obj->hello();
}
```

下面我们在扩展中实现以上test_call函数。

```
zend_class_entry *baby_ce;

ZEND_FUNCTION(test_call)
{
    zval *obj;
    MAKE_STD_ZVAL(obj);
    object_init_ex(obj, baby_ce);

    //如果确认此类没有构造函数就不用调用了。
    walu_call_user_function(NULL, obj, "__construct", "");

    walu_call_user_function(NULL, obj, "hello", "");
    zval_ptr_dtor(&obj);
    return;
}

ZEND_METHOD(baby, __construct)
{
    printf("a new baby!\n");
}

ZEND_METHOD(baby, hello)
{
    printf("hello world!!!!!\n");
}

static zend_function_entry baby_method[]={
    ZEND_ME(baby, __construct, NULL, ZEND_ACC_PUBLIC|ZEND_ACC_CTOR)
    ZEND_ME(baby, hello, NULL, ZEND_ACC_PUBLIC)
    {NULL, NULL, NULL}
};

ZEND_MINIT_FUNCTION(test)
{
    zend_class_entry ce;
    INIT_CLASS_ENTRY(ce, "baby", baby_method);
    baby_ce = zend_register_internal_class(&ce TSRMLS_CC);
    return SUCCESS;
}
```

links

- 11 PHP中的面向对象（二）
- 11.2 读写对象的属性

11.2 PHP中的面向对象（二）

在上一节里我们已经看了下如何操作一个对象的方法，这一节主要描述与对象属性有关的东西。有关如何对它进行定义的操作我们已经在上一章中描述过了，这里不再叙述，只讲对其的操作。

读取对象的属性

```
ZEND_API zval *zend_read_property(zend_class_entry *scope, zval *object, char *name, int

ZEND_API zval *zend_read_static_property(zend_class_entry *scope, char *name, int name_le
```

zend_read_property函数用于读取对象的属性，而zend_read_static_property则用于读取静态属性。可以看出，静态属性是直接保存在类上的，与具体的对象无关。 silent参数：

- 0: 如果属性不存在，则抛出一个notice错误。
- 1: 如果属性不存在，不报错。

如果所查的属性不存在，那么此函数将返回IS_NULL类型的zval。

更新对象的属性

```
ZEND_API void zend_update_property(zend_class_entry *scope, zval *object, char *name, int
ZEND_API int zend_update_static_property(zend_class_entry *scope, char *name, int name_le
```

zend_update_property用来更新对象的属性，zend_update_static_property用来更新类的静态属性。如果对象或者类中没有相关的属性，函数将自动的添加上。

读写对象与类属性的实例

假设我们已经在扩展中定义好下面的类：

```
class baby
{
    public $age;
    public static $area;

    public function __construct($age, $area)
    {
        $this->age = $age;
        self::$area = $area;

        var_dump($this->age, self::$area);
    }
}
```

```
ZEND_METHOD(baby, __construct)
{
    zval *age, *area;
    zend_class_entry *ce;
    ce = Z_OBJCE_P(getThis());
    if( zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &age, &area) == FAILURE )
    {
        printf("Error\n");
        RETURN_NULL();
    }
    zend_update_property(ce, getThis(), "age", sizeof("age")-1, age TSRMLS_CC);
    zend_update_static_property(ce, "area", sizeof("area")-1, area TSRMLS_CC);

    age = NULL;
    area = NULL;

    age = zend_read_property(ce, getThis(), "age", sizeof("age")-1, 0 TSRMLS_DC);
    php_var_dump(&age, 1 TSRMLS_CC);

    area = zend_read_static_property(ce, "area", sizeof("area")-1, 0 TSRMLS_DC);
    php_var_dump(&area, 1 TSRMLS_CC);

}
```

感谢 @看你取的破名 发现的错误。

一些其它的快捷函数

更新对象与类的属性

```
ZEND_API void zend_update_property_null(zend_class_entry *scope, zval *object, char *name
ZEND_API void zend_update_property_bool(zend_class_entry *scope, zval *object, char *name
ZEND_API void zend_update_property_long(zend_class_entry *scope, zval *object, char *name
ZEND_API void zend_update_property_double(zend_class_entry *scope, zval *object, char *na
ZEND_API void zend_update_property_string(zend_class_entry *scope, zval *object, char *na
ZEND_API void zend_update_property_stringl(zend_class_entry *scope, zval *object, char *n


ZEND_API int zend_update_static_property_null(zend_class_entry *scope, char *name, int na
ZEND_API int zend_update_static_property_bool(zend_class_entry *scope, char *name, int na
ZEND_API int zend_update_static_property_long(zend_class_entry *scope, char *name, int na
ZEND_API int zend_update_static_property_double(zend_class_entry *scope, char *name, int
ZEND_API int zend_update_static_property_string(zend_class_entry *scope, char *name, int
ZEND_API int zend_update_static_property_stringl(zend_class_entry *scope, char *name, int
```

links

- 11.1 生成对象的实例
- 11.3 小结

11.3 PHP中的面向对象（二）

有关面向对象的资料实在是太多了，我也是才学而已，没有办法给出非常系统、完整的阐述，但以后我会陆陆续续的在博客里写出来的。此外，强烈建议大家看看php官方的这篇wiki。 internals:engine:objects

links

- 11.2 读写对象的属性
- 12 启动与终止的那点事

12 启动与终止的那点事

目录

在前面的章节里，你已经学会了如何使用MINIT函数在PHP加载模块的共享库时来执行初始化任务。在第一章，你还了解到扩展里其他三个函数， 和MINIT函数对应的MSHUTDOWN函数，以及一对在每个页面请求开始和结束时候调用用的方法--RINIT函数和RSHUTDOWN函数。

links

- 目录
- 11.3 小结
- 12.1 关于生命周期

12.1 关于生命周期

除了在上一节说到的4个函数，还有2个函数只用于处理单个线程的启动和关闭，他们只作用于线程环境。

首先，建立一个基本扩展，根据你PHP源码树使用下面几个源文件。

config.m4

```
PHP_ARG_ENABLE(sample4,
    [Whether to enable the "sample4" extension],
    [ enable-sample4 Enable "sample4" extension support])
if test $PHP_SAMPLE4 != "no"; then
    PHP_SUBST(SAMPLE4_SHARED_LIBADD)
    PHP_NEW_EXTENSION(sample4, sample4.c, $ext_shared)
fi
```

php_sample4.h

```
#ifndef PHP_SAMPLE4_H
/* Prevent double inclusion */
#define PHP_SAMPLE4_H

/* Define Extension Properties */
#define PHP_SAMPLE4_EXTNAME
#define PHP_SAMPLE4_EXTVER

/* Import configure options when building outside of the PHP source tree */
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

/* Include PHP Standard Header */
#include "php.h"

/* Define the entry point symbol
 * Zend will use when loading this module
 */
extern zend_module_entry sample4_module_entry;
#define phpext_sample4_ptr &sample4_module_entry
#endif /* PHP_SAMPLE4_H */
```

sample4.c

```
#include "php_sample4.h"
#include "ext/standard/info.h"

static function_entry php_sample4_functions[] = {
    { NULL, NULL, NULL }
};

PHP_MINIT_FUNCTION(sample4)
```

```
{
    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(sample4) {
    return SUCCESS;
}

PHP_RINIT_FUNCTION(sample4) {
    return SUCCESS;
}
PHP_RSHUTDOWN_FUNCTION(sample4) {
    return SUCCESS;
}

PHP_MINFO_FUNCTION(sample4) {
}

zend_module_entry sample4_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE4_EXTNAME,
    php_sample4_functions,
    PHP_MINIT(sample4),
    PHP_MSHUTDOWN(sample4),
    PHP_RINIT(sample4),
    PHP_RSHUTDOWN(sample4),
    PHP_MINFO(sample4),
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE4_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_SAMPLE4
    ZEND_GET_MODULE(sample4)
#endif
```

注意：每个启动或者关闭的方法在return SUCCESS时退出。如果其中任何的函数return FAILURE，PHP为了避免出现严重问题而将请求中止。

现在你应该对MINIT很熟悉了吧，它会在一个模块第一次加载到进程空间的时候被触发。

对于多进程的SAPIS(Apache1 & Apache2-prefork)，多个web server进程会fork出多个mod_php实例。每个mod_php实例都必须加载属于这个实例 的扩展模块，这意味着MINIT函数会被执行多次。但是，它在每个进程空间中只会执行一次。

当一个模块被卸载，MSHUTDOWN会被调用，它可以使用该模块的任何资源，比如被占用的内存可能会被释放。

这里要注意个特性, 某些PHP的SAPI中, 比如Apache Prefork, PHP是作为一个动态库被加载到Apache中的, 而从Apache 1.3以后(如果我没记错的话), Apache做了一个优化, 优化的结果就是首先执行各个动态模块的模块初始化工作, 然后才做fork, 派生Worker子进程, 所以反应到这里, 有的时候会出现MINIT只执行一次, 而MSHUTDOWN会执行多次的现象.

理论上来说，你可以在MSHUTDOWN中跳过一些资源的清理工作，然而在APACHE 1.3上的时候，你会发现一个有趣的事情，apache会载入mod_php， 并且会执行所有的MINIT方法，然后立刻卸载mod_php来触发MSHUTDOWN，接着再次装入，在没有执行MSHUTDOWN的时候，最初使用MINIT加载的 资源将被泄露和浪费。

在多线程的SAPIS中，有时需要为每个线程分配自己独立的资源或跟踪每个请求的计数器。对于这些特殊情况，在每一个线程钩子中，允许额外的启动和关闭 要执行的方法。通常情况下，当多进程的SAPIS(Apache2-worker)启动时，它会创建出十几个或者更多的线程，以便能够处理多个并发请求。

任何可以请求之间共享，但不能由多个线程在同一进程空间同时访问的资源，通常分配在线程的构造和析构方法中以免发生冲突。比如可能包括在 EG（ persistent_list ） HashTable中的持久性资源，因为他们往往包括网络或文件资源。

links

- 目录
- 上一节: 启动与终止的那点事
- 下一节: MINFO与phpinfo

12.2 MINFO与phpinfo

如果你并不打算做出一个只有你自己使用的扩展，那么你可以需要告诉用户一些关于你的扩展信息。比如：其环境和特定版本的可用功能、版本信息、 作者信息以便你在发生问题的时候可以寻求帮助、甚至可以加上一个LOGO等等。

如果你仔细看过phpinfo()或者 php -i 的输出，相信你已经注意到，所有这些信息会组合成一个格式良好的、易于解析输出。 你的扩展可以轻松地通过 MINFO (模块信息)来添加这些块，看个例子：

```
PHP_MINFO_FUNCTION(sample4) {
    php_info_print_table_start();
    php_info_print_table_row(2, "Sample4 Module", "enabled");
    php_info_print_table_row(2, "version", PHP_SAMPLE4_EXTVER);
    php_info_print_table_end();
}
```

通过使用这些包装的功能，你的模块的信息将被自动包裹在HTML标签中从一个网络服务器SAPI （如CGI时，IIS， APACHE，等等）输出，或格式化使 用CLI使用时，输出明文和换行符。

下面我们来介绍一下php*info**()系列的函数：

```
char *php_info_html_esc(char *str TSRMLS_DC)
```

这个函数是php_escape_html_entities()的一个封装，htmlentites() 函数的底层实现。该函数返回的字符串通过emalloc()创建，并在使用后必须使用 efree()函数释放掉。

```
void php_info_print_table_start(void)
void php_info_print_table_end(void)
```

输出开/关表格式所需的标签。HTML输出是与CLI输出一样，表现为一个简单的换行。

```
void php_info_print_table_header(int cols, ...)
void php_info_print_table_colspan_header(int cols, char *header)
```

输出表头行。第一个函数在可变参数列表中的char *元素外面的每一列都会输出一对th标签，第二个函数会在指定列数外面输出一对th标签。

```
void php_info_print_table_row(int cols, ...)
void php_info_print_table_row_ex(int cols, char *class, ...)
```

第一个函数在可变参数列表中的char *元素外面的每一行都会输出一对td标签，第二个函数会在指定列数外面输出一对td标签。当不在HTML中 输出的时候，两个函数将没有任何差别。

```
void php_info_print_hr(void)
```

这种函数将在HTML中输出一个br标签，或者一个表示行开始和结束的水平线

我们常用的PHPWRITE()和php_printf()函数可以在在MINFO函数中使用，你应该注意正确的信息输出取决于当前的SAPI判断是用纯文本还是HTML的方式输出 要做到这一点，只需要检查sapi_module结构中的phpinfo_as_text属性，例子如下：

```
PHP_MINFO_FUNCTION(sample4) {
    php_info_print_table_start();
    php_info_print_table_row(2, "Sample4 Module", "enabled");
    php_info_print_table_row(2, "version", PHP_SAMPLE4_EXTVER);
    if (sapi_module.phpinfo_as_text) {
        /* No HTML for you */
        php_info_print_table_row(2, "By",
            "Example Technologies\nhttp://www.example.com");
    } else {
        /* HTMLified version */
        php_printf("<tr>"
            "<td class=\"v\">By</td>"
            "<td class=\"v\">"
            "<a href=\"http://www.example.com\""
            " alt=\"Example Technologies\">"
            "<img src=\"http://www.example.com/logo.png\" />"
            "</a></td></tr>");
        php_info_print_table_end();
    }
}
```

links

- 目录
- 12.1 关于生命周期
- 12.3 常量

12.3 常量

在脚本中使用扩展的一个方便之处是，人们可以改变自己定义的常量。你可以通过define()函数来定义一个常量。在内核中，我们将会使用REGISTER_*_CONSTANT()的 家族函数来使用常量。

对于你定义的大多数常量来说，你可能希望在程序初始化的时候便定义这些变量。你可能需要在MINIT函数：

```
PHP_MINIT_FUNCTION(sample4) {
    REGISTER_STRING_CONSTANT("SAMPLE4_VERSION",
            PHP_SAMPLE4_EXTVER, CONST_CS | CONST_PERSISTENT);
    return SUCCESS;
}
```

第一个参数是你要定义的这个常量的名字。在例子中，我们定义了一个名称为SAMPLE4*VERSION*的常量。有一点很重要，这里要注意宏*REGISTER_CONSTANT()*的 使用，这些函数中为了确定常量的名称长度使用了*sizeof()*。这就意味着，常量的名称只能为文字，大家可以尝试使用一个*char* 的变量，这将导致sizeof计算出错误 的字符串长度。

接下来，我们来看看常量的值。在大多数情况下，它会是一个单一参数的类型，然而在STRINGL的版本中，你会看到在一些情况下会需要使用第二个参数来表明长度。 当注册string类型的常量时，字符串的值不会被复制到常量中，而仅仅是一个引用。这意味着，动态创建的字符串需要持久化和在shutdown的阶段被释放掉。

最后，在最后一个参数，你可以通过两个可以标识位的按位或组合传入。CONST_CS标识是否大小写敏感，一般情况下CONST_CS标识是默认使用的。对于一些特殊的 情况，比如TRUE,FALSE,NULL等等，这个参数将被省略。

在|后的标识位中的标识符说明了该常量的作用域和生命周期。当我们在MINIT中定义常量时，你可能需要在多个请求中使用这个常量，当你在RINIT中定义常量时，这个 常量会在当前请求结束的时候销毁。

下面列出的4个创建常量常用的函数，有一个共同需要注意的地方，常量名称一定要用文字而不是char *类型的变量。

```
REGISTER_LONG_CONSTANT(char *name, long lval, int flags)
REGISTER_DOUBLE_CONSTANT(char *name, double dval, int flags)
REGISTER_STRING_CONSTANT(char *name, char *value, int flags)
REGISTER_STRINGL_CONSTANT(char *name,char *value, int value_len, int flags)
```

如果你没有办法提供文本类型的name，那么你可以尝试使用上面4个函数的底层函数去实现相同的效果：

```
void zend_register_long_constant(char *name, uint name_len, long lval, int flags, int mod
void zend_register_double_constant(char *name, uint name_len, double dval, int flags, int
void zend_register_string_constant(char *name, uint name_len, char *strval, int flags, in
void zend_register_stringl_constant(char *name, uint name_len, char *strval, uint strlen,
```

这样就可以由传入name_len而扩大了该族函数的使用范围(比如在循环中)。

module_number是一个加载扩展或者卸载扩展时的标识。而你不需要关注它，它会自动加载到你扩展中的MINIT和RINIT中，所以在你用上面4个函数声明常量的时候， 你可以这样写：

```
PHP_MINIT_FUNCTION(sample4) {
    register_string_constant("SAMPLE4_VERSION",
                    sizeof("SAMPLE4_VERSION"),
                    PHP_SAMPLE4_EXTVER,
                    CONST_CS | CONST_PERSISTENT,
                    module_number TSRMLS_CC);
    return SUCCESS;
}
```

除了数组和对象外，其他变量你也可以用来注册一个常量，但是因为没有宏和ZEND API去支持这些声明，所以你必须手动声明一个常量，通过下面一个例子来了解一下：

```
void php_sample4_register_boolean_constant(char *name, uint len,
            zend_bool bval, int flags, int module_number TSRMLS_DC)
{
        zend_constant c;

        ZVAL_BOOL(&c.value, bval);
        c.flags = CONST_CS | CONST_PERSISTENT;
        c.name = zend_strndup(name, len - 1);
        c.name_len = len;
        c.module_number = module_number;
        zend_register_constant(&c TSRMLS_CC);
}
```

links

* 目录
* 12.2 MINFO与phpinfo
* 12.4 PHP扩展中的全局变量

12.4 PHP扩展中的全局变量

这一章，我们将学会如何在PHP扩展中使用全局变量。

在扩展中定义全局变量

首先，我们需要在扩展的头文件中(默认是php_*.h)中定义所有的全局变量。举个例子，比如我们要定义一个无符号的long类型的全局变量，我们可以这样定义：

```
ZEND_BEGIN_MODULE_GLOBALS(sample4)
    unsigned long counter;
ZEND_END_MODULE_GLOBALS(sample4)
```

用ZEND_BEGIN_MODULE_GLOBALS和ZEND_END_MODULE_GLOBALS宏将定义的全局变量包起来。将上例中的宏展开后，是下面这个样子：

```
typedef struct _zend_sample4_globals {
    unsigned long counter;
} zend_sample4_globals;
```

如果你还有其他的全局变量需要定义，只需加在两个宏之间就可以了。接下来我们该在simple4.c中声明我们在头文件中定义的这些全局变量了：

```
ZEND_DECLARE_MODULE_GLOBALS(sample4);
```

这个宏的内部实现取决于是否启用了线程安全，在非线程安全的环境下，如：Apache1，Apache2-prefork，CGI，CLI...会使用zend_sample4_globals结构来定义 全局变量：

```
zend_sample4_globals sample4_globals;
```

我们可以直接通过sample4_globals.counter来获取计数器的值。在线程安全的版本中，另一种方法是声明一个整数：

```
int sample4_globals_id;
```

填充这个ID就等于定义了扩展中的全局变量。根据其定义的信息，将为每个新线程的独立存储空间分配内存块。我们可以在MINIT中这样定义：

```
#ifdef ZTS
    ts_allocate_id(
            &sample4_globals_id,
            sizeof(zend_sample4_globals),
            NULL, NULL);
```

```
#endif
```

有一点需要注意这种方法需要包裹在#ifdef中，以防止它在没有启动Zend Thread Safety(ZTS)时执行。因为sample4_globals_id只能在多线程环境中使用。非线程 的版本用我们在前面提到的sample4_globals来声明全局变量。

线程中的初始化和关闭

在非线程的环境中，会将一个zend_sample4_globals结构的副本保存在指定进程中。你可以指定他的默认值，或者在MINIT或者RINIT中分配资源来初始化它。要记得 在对应的MSHUTDOWN或者RSHUTDOWN中及时释放这些资源。

然而在线程版本中，一个新的结构会在一个新线程spun的时候被分配。为了知道怎样初始化和关闭扩展中的全局变量，需要向ZE引擎提供回调函数。在前面我们在调用 ts_allocate_id()的时候是是以NULL来作为这个值的，接下来我们添加2个一会需要在MINIT调用的方法：

```
static void php_sample4_globals_ctor(zend_sample4_globals *sample4_globals TSRMLS_DC)
{
    /* Initialize a new zend_sample4_globals struct
     * During thread spin-up */
     sample4_globals->counter = 0;
}

static void php_sample4_globals_dtor(zend_sample4_globals *sample4_globals TSRMLS_DC)
{
    /* Any resources allocated during initialization
     * May be freed here */
}
```

我们在启用和关闭的时候调用它们：

```
PHP_MINIT_FUNCTION(sample4) {
    REGISTER_STRING_CONSTANT("SAMPLE4_VERSION", PHP_SAMPLE4_EXTVER, CONST_CS | CONST_PERS
#ifdef ZTS
    ts_allocate_id(&sample4_globals_id,
                        sizeof(zend_sample4_globals),
                        (ts_allocate_ctor)php_sample4_globals_ctor,
                        (ts_allocate_dtor)php_sample4_globals_dtor);
#else
    php_sample4_globals_ctor(&sample4_globals TSRMLS_CC);
#endif
    return SUCCESS;
}

PHP_MSHUTDOWN_FUNCTION(sample4) {
#ifndef ZTS
    php_sample4_globals_dtor(&sample4_globals TSRMLS_CC);
#endif
    return SUCCESS;
}
```

现在我们已经知道如何在扩展中创建全局变量了，在不是ZTS的环境中，使用它们很简单，我们还来看前面定义的那个计数器的递增功能如何实现：

```
PHP_FUNCTION(sample4_counter) {
    RETURN_LONG(++sample4_globals.counter);
}
```

是不是看起来很简单，但是，在线程版本中将无法正常工作。那么我们来看看怎么在线程环境中完成这个功能吧：

```
PHP_FUNCTION(sample4_counter)
{
#ifdef ZTS
        RETURN_LONG(++TSRMG(sample4_globals_id, \
                zend_sample4_globals*, counter));
#else
        /* non-ZTS */
        RETURN_LONG(++sample4_globals.counter);
#endif
}
```

看起来很丑对吗？想象一下，在你的整个代码库中，这些IFDEF指令在每一个线程安全的全局访问时穿插。它看起来比Perl更糟糕！这就是为什么所有 的核心扩展，都有使用一个额外的宏观层抽象这种情况。我们可以在php_sample4.h中找到下面这段代码：

```
#ifdef ZTS
#include "TSRM.h"
#define SAMPLE4_G(v) TSRMG(sample4_globals_id, zend_sample4_globals*, v)
#else
#define SAMPLE4_G(v) (sample4_globals.v)
#endif
```

使用它们会让你的方法看起来更简洁：

```
PHP_FUNCTION(sample4_counter) {
    RETURN_LONG(++SAMPLE4_G(counter));
}
```

看到*G()这样的宏是不是有种似曾相识的感觉？也许以前你看到过EG()、CG()等宏，了解他们会让你对PHP的了解更深一步：

Accessor Macro	Associated Data
EG()	这个宏可以用来访问符号表，函数，资源信息和常量。
CG()	用来访问核心全局变量。
PG()	PHP全局变量。我们知道php.ini会映射一个或者多个PHP全局结构。举几个使用这个宏的例子：PG(register_globals), PG(safe_mode), PG(memory_limit)

FG()	文件全局变量。大多数文件I/O或相关的全局变量的数据流都塞进标准扩展出口结构。

links

- 目录
- 12.3 常量
- 12.5 PHP语言中的超级全局变量

12.5 PHP语言中的超级全局变量(Superglobals)

在PHP中有一种“特殊”的全局变量，通常我们把它们称作超级全局变量，常见的比如 `$_GET`、`$_POST`、`$_FILE` 等等。

他们会在编译之前就声明，所以在普通的脚本中，可能无法定义其它的超级全局变量。在扩展中，最好的使用超级全局变量的是session扩展，它使用 `$_SESSION` 来在 `session_start()` 和 `session_write_close()` 之间存储信息。那么是怎样定义 `$_SESSION` 这个超级全局变量的呢？我们来看下session扩展的 `MINIT` 函数实现：

```
PHP_MINIT_FUNCTION(session) {
        zend_register_auto_global("_SESSION",
                                  sizeof("_SESSION") - 1,
                                  NULL TSRMLS_CC);
        return SUCCESS;
}
```

注意这里的第二个参数，`sizeof("_SESSION") - 1`，一定要排除标识字符串结束的\0符。

我们一起来看下 `zend_register_auto_global()` 这个函数在ZE2中的原型：

```
int zend_register_auto_global(char *name, uint name_len,
            zend_auto_global_callback auto_global_callback TSRMLS_DC)
```

在ZE1中，是没有 `auto_global_callback` 这个参数的。为了和PHP4兼容，我们仍需要像下面这样声明一个超级全局变量：

```
PHP_MINIT_FUNCTION(sample4) {
        zend_register_auto_global("_SAMPLE4", sizeof("_SAMPLE4") - 1
#ifdef ZEND_ENGINE_2
                , NULL
#endif
                TSRMLS_CC);
        return SUCCESS;
}
```

全局变量的回调

在ZE2中，`zend_register_auto_global()` 函数的 `auto_global_callback` 参数接受一个自定义函数。在实践中，这样的做法可以用来避免复杂的初始化，我们来 看下面这一段代码：

```
zend_bool php_sample4_autoglobal_callback(char *name, uint name_len TSRMLS_DC)
{
    zval *sample4_val;
    int i;
    MAKE_STD_ZVAL(sample4_val);
```

```
    array_init(sample4_val);
    for(i = 0; i < 10000; i++) {
        add_next_index_long(sample4_val, i);
    }
    ZEND_SET_SYMBOL(&EG(symbol_table), "_SAMPLE4", sample4_val);
    return 0;
}

PHP_MINIT_FUNCTION(sample4) {
    zend_register_auto_global("_SAMPLE4", sizeof("_SAMPLE4") - 1
#ifdef ZEND_ENGINE_2
                        , php_sample4_autoglobal_callback
#endif
                        TSRMLS_CC);
    return SUCCESS;
}
```

不幸的是，这样的设计打破了PHP4和ZE1的规则，它们不支持这样的回调。所以，为了兼容它们，我们要在每个脚本开始的时候去调用我们的回调函数(RINIT)：

```
PHP_RINIT_FUNCTION(sample4) {
#ifdef ZEND_ENGINE_2
    php_sample4_autoglobal_callback("_SAMPLE4",
                            sizeof("_SAMPLE4") - 1,
                            TSRMLS_CC);
#endif
    return SUCCESS;
}
```

links

- 目录
- 12.4 PHP扩展中的全局变量
- 12.6 小结

12.6 小结

通过本章的课程，我们深入了解了PHP的生命周期，常量、全局变量和超级全局变量的定义和使用。在下一章中，你会学会如何声明和使用的php.ini值。

links

- 目录
- 12.5 PHP语言中的超级全局变量
- 13 设置INI

13 INI设置

目录

在前面的一章，我们已经学会了MINIT、MSHUTDOWN函数，以及RINIT和RSHUTDOWN等函数的使用，这里我们将介绍并学习ini设置的使用。

links

- 目录
- 12.6 小结
- 13.1 声明和访问INI设置

13.1 声明和访问INI设置

INI条目被定义在一个完整的独立的的块，位于上文中所说的MINIT方法的同一个源文件，并且用下面的一对宏来定义，并在这对宏之间放入一个或者多个条目 `PHP_INI_BEGIN()` 和 `PHP_INI_END()`

这些宏方法和上一章所提到的 `ZEND_BEGIN_MODULE_GLOBALS()` 和 `ZEND_END_MODULE_GLOBALS()` 有着相同的用法。这些结构是用静态数据的实例来声明，而不仅仅是提供一个结构的定义

```
static zend_ini_entry ini_entries[] = {
           {0,0,NULL,0,NULL,NULL,NULL,NULL,NULL,0,NULL,0,0,NULL} };
```

正如你所看到的，上面定义了 `zend_ini_entry` 的一个向量值并以一个空记录来终止。你或许已经多次在 `function_entry` 结构的定义之中看到过这种以填充静态向量的方法

简单INI设置

现在你可以使用 INI 结构来声明条目，这个机制是用来注册和销毁一些在机器上的设置，你可以声明一些对你的扩展有用的有实际意义的设置了。

假设你的扩展的方法就像你在第五章看到的那个例子（ `Your First Extension` ）一样，只是输出一个简单的问候，你可能想让你要输出的问候语句是可定制的：

```
PHP_FUNCTION(sample4_hello_world)
{
    php_printf("Hello World!\n");
}
```

最简单的方法就是定义一个 INI 的设置，让它的默认值为 `Hello World!` ，像下面这样：

```
#include "php_ini.h"
PHP_INI_BEGIN()
    PHP_INI_ENTRY("sample4.greeting", "Hello World", PHP_INI_ALL, NULL)
PHP_INI_END()
```

正如你猜测的一样，PHP_INI_ENTRY 这个宏里面设置的前面的两个参数，分别代表着INI设置的名称和它的默认值。第二个参数决定设置是否允许被修改，以及它能被修改的作用域。最后一个参数是一个回调函数，当INI的值被修改时候触发此回调函数。你将会在某些修改事件的地方详细的了解这个参数。

PHP总共有4个指令配置作用域：（PHP中的每个指令都有自己的作用域，指令只能在其作用域中修改，不是任何地方都能修改配置指令的）

Parameter	Meaning
PHP_INI_PERDIR	指令可以在php.ini、httpd.conf或.htaccess文件中修改
PHP_INI_SYSTEM	指令可以在php.ini 和 httpd.conf 文件中修改

PHP_INI_USER	指令可以在用户脚本中修改
PHP_INI_ALL	指令可以在任何地方修改

现在你已经声明了你的INI的设置，你现在准备将它用在你的问候函数之中：

```
PHP_FUNCTION(sample4_hello_world)
{
    const char *greeting = INI_STR("sample4.greeting");
    php_printf("%s\n", greeting);
}
```

有一点很重要：`char*` 类型的值被认为是属于 `ZEND` 引擎的，是不能修改的。正因为如此，你将本地变量设置进INI时候，它将在你的方法中被声明为 `const` 。并不是所有的INI值都是基于字符串的；也有其他的一些用于整数、浮点数、或布尔值的宏，例如：

```
long lval = INI_INT("sample4.intval");
double dval = INI_FLT("sample4.fltval");
zend_bool bval = INI_BOOL("sample4.boolval");
```

通常你想知道你当前的 `INI` 设置的值；恭喜你，ZEND内核刚好就存在一组这样的宏为你提供查询每种类型的INI的默认值。

```
const char *strval = INI_ORIG_STR("sample4.stringval");
long lval = INI_ORIG_INT("sample4.intval");
double dval = INI_ORIG_FLT("sample4.fltval");
zend_bool bval = INI_ORIG_BOOL("sample4.boolval");
```

NOTE

```
在这个例子中，`sample4.greeting`只是INI设置的一个前缀，用来帮助保证它的设置不会和其他扩展的设置相冲突
```

访问级别

每一个给出的INI值总是有默认值的。在许多情况下，INI都有一个比较合理的默认值；然而，在某些比较特殊的环境或者某个脚本要做一些比较特别的事情的情况下，这些INI值通常会被修改。这些设置可以在任何的三个不同的点被修改，看下表：

Access Level	Meaning
SYSTEM	设置被放在php.ini中，或者在Apache的http.conf配置文件中的和，它在apache启动时候生效，被认为是设置的全局变量
PERDIR	一些设置被放在Apache的http.conf的或者块中，或者.htaccess文件之中。原文：Any setting found in a or block within Apache's httpd.conf, or settings located in .htaccess files as well as certain other locations not exclusive to Apache are processed just prior to a given request if that request is within the appropriate directory or virtual host.
USER	一旦脚本开始执行，唯一的改变INI设置的方法就是利用用户方法：ini_set()

某些设置，例如safe_mode，如果他们能在任何地方任意修改的话，那它的存在就没意义了。例如，一个恶意脚本的作者可以简单的禁用safe_mode,然后任意读取和修改其他的被禁止的文件。

同样，一些非安全相关的设置，如：register_globals或magic_quotes_gpc 不能在一个脚本中有效的被改变，因为他们所承担的任务已经过了。 Similarly, some non-security related settings such as register_globals or magic_quotes_gpc cannot be effectively changed within a script because the point at which they bear relevance has already passed.

这些设置是通过 `PHP_INI_ENTRY()` 的第三个参数来设置的。在你的设置声明之中，你已经使用了 `PHP_INI_ALL` , 他们是按位或者 `PHP_INI_SYSTEM | PHP_INI_PERDIR | PHP_INI_USER` 的组合来定义的。

如register_globals和magic_quotes_gpc的设置，反过来，使用 `PHP_INI_SYSTEM | PHP_INI_PERDIR` 来声明。对于这些设置，在任何调用 `ini_set()` 的地方排除 `PHP_INI_USER` 的话将以失败告终（The exclusion of PHP_INI_USER results in any call to ini_set() for these settings ending in failure）。

现在你可能会猜测，safe_mode和open_basedir之类的设置只能用 `PHP_INI_SYSTEM` 来声明。这种设置可以确保只有系统管理员才能修改这些值，因为只有他们有权限修改php.ini或者http.conf中的值。

修改事件

无论INI设置在什么时候被修改，无论是通过 `ini_set()` 方法来修改还是在一个 `perdir` 指令执行期间来修改，zend引擎都会通过一个 `OnModify` 的回调来检查它。做修改的人可能会通过使用 `ZEND_INI_MH` 宏来定义，然后通过 `OnModify` 方法的参数来附加到INI设置里面：

```
ZEND_INI_MH(php_sample4_modify_greeting)
{
    if (new_value_length == 0) {
        return FAILURE;
    }
    return SUCCESS;
}
PHP_INI_BEGIN()
    PHP_INI_ENTRY("sample4.greeting", "Hello World",
        PHP_INI_ALL, php_sample4_modify_greeting)
PHP_INI_END()
```

当 `new_value_length` 的长度为0的时候返回FAILURE，这样修改者就可以禁止将祝福语句设置为一个空字符串。像下面这样使用 `ZEND_INI_MH()` 可以生成整个原型：

```
int php_sample4_modify_greeting(zend_ini_entry *entry,
          char *new_value, uint new_value_length,
          void *mh_arg1, void *mh_arg2, void *mh_arg3,
          int stage TSRMLS_DC);
```

Table 13.2. INI Setting Modifier Callback Parameters

Parameter	Meaning
enTRy	指向zend引擎实际存储的INI设置，这种结构提供一些信息，包括现在的值、原始值、拥有模块、还有其他的一些在下面的 Listing 13.1 中的详细信息

new_value	关于设置的值。如果处理方法返回SUCCESS，这个值将被填充进 enTRy->value，如果 entry->orig_value 至今没有设置的话，当前的值将被旋转到这个位置，并且也会设置 enTRy->modified 这个标志。字符串的长度将被填充到 new_value_length。
mh_arg1,2,3	这组指针（三个一组）提供了访问数据指针最初给出的INI设置的声明。在实践中，这些值都是通过内部引擎进程来调用的,所以你不需要担心它们
stage	ZEND_INI_STAGE_s 这种形式里面有五个值，这五个由s代表的值为STARTUP, SHUTDOWN, ACTIVATE, DEACTIVATE, 或者 RUNTIME，这些常量分别对应 MINIT, MSHUTDOWN, RINIT, RSHUTDOWN 还有 处于活跃状态正在执行的脚本。

Listing 13.1. Core structure: zend_ini_entry

```
struct _zend_ini_entry {
    int module_number;
    int modifiable;
    char *name;
    uint name_length;
    ZEND_INI_MH((*on_modify));
    void *mh_arg1;
    void *mh_arg2;
    void *mh_arg3;
    char *value;
    uint value_length;
    char *orig_value;
    uint orig_value_length;
    int modified;
    void ZEND_INI_DISP(*displayer);
};
```

显示INI设置

在上一章，你已经见过 `MINFO` 方法，还有一些其他关于扩展的基础知识。因为输出INI的信息对于一个扩展来说是很正常的，ZEND引擎有一个统一的宏来输出这些内容，它可以被放置在 `PHP_MINFO_FUNCTION()` 块中：

```
PHP_MINFO_FUNCTION(sample4)
{
    DISPLAY_INI_ENTRIES();
}
```

这个宏需要INI设置的值，而这个值早已经在前面定义在了 `PHP_INI_BEGIN` 和 `PHP_INI_END` 宏之间。INI设置被迭代的显示在一个三列的表单中，这三列分别是INI设置的名字，它的原始（全局）设置，还有通过 `PERDIR` 指令或者 `ini_set()` 修改过的当前的设置。

默认情况下，所有的设置的条目都是根据原有的字符串来简单的输出的。(By default, all entries are simply output according to their string representation as-is.)一些类似布尔值和语法高亮显示的颜色的值，在显示的时候会经过特殊的格式化处理。这种特殊格式化的方法是通过每个INI设置自己的显示处理程序来处理的，这个处理程序是通过动态指向一个回调函数来实现的，跟我们前面看到的 OnModify 类似。

显示处理程序指定使用一个扩展的 `PHP_INI_ENTRY()` 宏的版本，`PHP_INI_ENTRY()` 接受一个额外的参数。如果将它设置为NULL，默认的显示处理程序将按照字符串的值原样使用。

```
PHP_INI_ENTRY_EX("sample4.greeting", "Hello World", PHP_INI_ALL,
    php_sample4_modify_greeting, php_sample4_display_greeting)
```

明显的，这个回调需要在INI设置使用之前提前的定义好。与 OnModify 的回调类似，这个用一个包装宏来完成，并且只用少量的代码就能实现：

```
#include "SAPI.h" /* needed for sapi_module */
PHP_INI_DISP(php_sample4_display_greeting)
{
    const char *value = ini_entry->value;
    /* Select the current or original value as appropriate */
    if (type == ZEND_INI_DISPLAY_ORIG &&
        ini_entry->modified)  {
        value = ini_entry->orig_value;
    }
    /* Make the greeting bold (when HTML output is enabled) */
    if (sapi_module.phpinfo_as_text) {
        php_printf("%s", value);
    } else {
        php_printf("<b>%s</b>", value);
    }
}
```

绑定到扩展的全局设置

所有的INI条目都在Zend Engine中被给予了存储空间，用来在脚本之中追踪它的改变，并且在请求之外维持全局设置。在这个存储空间之中，所有的INI设置都是以字符串形式保存的。你应该会想到，这些值可以使用 `INI_INT()` , `INI_FLT()` ,和 `INI_BOOL()` 这类宏很容易地转换为标量值。

这个查询和转换过程非常低效的原因有两个，它必须通过名字位于一个hash表之中，所以每次都要重新获取。这种查找方式对于用户空间脚本是非常好的，因为一个脚本只有在运行时才会被编译，但是对于编译型的语言,做这个工作是毫无意义的。

对于标量来说，这个会导致更加的低效，因为底层的字符串值在每一次被请求时候都必须再转换一次。使用你已经知道的，你可以声明一个线程安全的全局存储介质，并在每次改变的时候使用新值的地址来更新它。然后，任何代码都可以通过在你的线程安全的结构之中查找指针来访问INI设置，这个可以利用编译时优化。

在php_sample4.h文件中，在 `MODULE_GLOBALS` 结构中添加 `const char *greeting;` ，然后更新位于sample4.c中的两个方法：

```
ZEND_INI_MH(php_sample4_modify_greeting)
{
    /* Disallow empty greetings */
    if (new_value_length == 0) {
        return FAILURE;
    }
    SAMPLE4_G(greeting) = new_value;
    return SUCCESS;
}
PHP_FUNCTION(sample4_hello_world)
```

```
{
    php_printf("%s\n", SAMPLE4_G(greeting));
}
```

因为这是一个优化 INI 值存取的非常一般的方法，另一对宏是通过Zend引擎来导出的，将INI设置绑定到全局变量。(Because this is a common approach to optimizing INI access, another pair of macros is exported by the engine that handle binding INI settings to global variables)

```
STD_PHP_INI_ENTRY_EX("sample4.greeting", "Hello World",
    PHP_INI_ALL, OnUpdateStringUnempty, greeting,
    zend_sample4_globals, sample4_globals,
    php_sample4_display_greeting)
```

这个条目和刚刚你不需要的回调执行相同的工作。相反，他使用一个通用目的的修饰回调 `OnUpdateStringUnempty`，并且连同信息在存储空间的存储位置。为了允许空白的问候语，你可以简单的指定 `OnUpdateString` 修饰语，这样就比 `OnUpdateStringUnempty` 方法简单。

类似的方法，比如INI设置可能会被绑定到类似long,double和zend_bool之类的标量上。在你的php_sample4.h中添加三个条目到MODULE_GLOBALS 结构中：

```
long mylong;
double mydouble;
zend_bool mybool;
```

现在使用 `STD_PHP_INI_ENTRY()` 宏在 `PHP_INI_BEGIN()/PHP_INI_END()` 块中创建INI条目，不同于它的 `_EX` 版本的仅仅是缺少一个播放者方法以及绑定他们到你的新值：

```
STD_PHP_INI_ENTRY("sample4.longval", "123",
    PHP_INI_ALL, OnUpdateLong, mylong,
    zend_sample4_globals, sample4_globals)
STD_PHP_INI_ENTRY("sample4.doubleval", "123.456",
    PHP_INI_ALL, OnUpdateDouble, mydouble,
    zend_sample4_globals, sample4_globals)
STD_PHP_INI_ENTRY("sample4.boolval", "1",
    PHP_INI_ALL, OnUpdateBool, mybool,
    zend_sample4_globals, sample4_globals)
```

注意在这一点上，如果 `DISPLAY_INI_ENTRIES()` 被调用，"sample4.boolval" 这个布尔类型的INI设置会像其他的设置一样以字符串被显示出来；然而，优先以字符串输出的是"on"或者“off”。为了确认这些显示的值是有意义的，你可以在后面的两个方法之中任选一个，其中一个是切换到 `STD_PHP_INI_ENTRY_EX()` 宏创建一个显示的方法，另一个是你可以任意使用可以帮到你的宏：

```
STD_PHP_INI_BOOLEAN("sample4.boolval", "1",
    PHP_INI_ALL, OnUpdateBool, mybool,
    zend_sample4_globals *, sample4_globals)
```

这种特定类型的宏跟在INI家族中的布尔是不一样的，它仅仅提供一个显示处理者来将开启的设置为“on”,

关闭的设置为"off"。

links

- 目录
- 13 INI设置
- 13.2 小结

13.2 小结

在这章，你探索了PHP中最古老的特性，也可以说是PHP健壮可移植性的最大的问题所在。对于每个有用的INI设置，它对编程的障碍就是随时都会出现它，这让编程越来越复杂。INI设置是一把双刃剑，所以在使用之前一定要深思熟虑，不然造成的后果将是长久的；胡乱使用将使你的系统出现很多不可预测的问题。

在接下来的三章中，你将专研数据流API，从使用开始，进一步提升到数据流的实现层,包装、环境以及过滤器

links

14 流式访问

目录

PHP用户空间中所有的文件I/O处理都是通过php 4.3引入的php流包装层处理的。在内部，扩展代码可以选择使用stdio或posix文件处理和本地文件系统或伯克利域套接字进行通信，或者也可以调用和用户空间流I/O相同的API。

links

- 目录
- 12.6 小结
- 14.1 概览

14.1 流的概览

通常, 直接的文件描述符相比调用流包装层消耗更少的CPU和内存; 然而, 这样会将实现某个特定协议的所有工作都堆积到作为扩展开发者的你身上. 通过挂钩到流包装层, 你的扩展代码可以透明的使用各种内建的流包装, 比如HTTP, FTP, 以及它们对应的SSL版本, 另外还有gzip和bzip2压缩包装. 通过include特定的PEAR或PECL模块, 你的代码还可以访问其他协议, 比如SSH2, WebDav, 甚至是Gopher!

本章将介绍内部基于流工作的基础API. 后面到第16章"有趣的流"中, 我们将看到诸如应用过滤器, 使用上下文选项和参数等高级概念.

打开流

尽管是一个统一的API, 但实际上依赖于所需的流的类型, 有四种不同的路径去打开一个流. 从用户空间角度来看, 这四种不同的类别如下(函数列表只代表示例, 不是完整列表):

```
<?php
/* fopen包装
* 操作文件/URI方式指定远程文件类资源 */
$fp = fopen($url, $mode);
$data = file_get_contents($url);
file_put_contents($url, $data);
$lines = file($url);
/* 传输
* 基于套接字的顺序I/O */
$fp = fsockopen($host, $port);
$fp = stream_socket_client($uri);
$fp = stream_socket_server($uri, $options);

/* 目录流 */
$dir = opendir($url);
$files = scandir($url);
$obj = dir($url);

/* "特殊"的流 */
$fp = tmpfile();
$fp = popen($cmd);
proc_open($cmd, $pipes);
```

无论你打开的是什么类型的流, 它们都存储在一个公共的结构体php_stream中.

fopen包装

我们首先从实现fopen()函数开始. 现在你应该已经对创建扩展骨架很熟悉了, 如果还不熟悉, 请回到第5章"你的第一个扩展"复习一下, 下面是我们实现的fopen()函数:

```
PHP_FUNCTION(sample5_fopen)
{
    php_stream *stream;
```

```
    char *path, *mode;
    int path_len, mode_len;
    int options = ENFORCE_SAFE_MODE | REPORT_ERRORS;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "ss",
        &path, &path_len, &mode, &mode_len) == FAILURE) {
        return;
    }
    stream = php_stream_open_wrapper(path, mode, options, NULL);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

php_stream_open_wrapper()的目的应该是完全绕过底层. path指定要读写文件名或URL, 读写行为依赖于mode的值.

options是位域的标记值集合, 这里是设置为下面介绍的一组固定值:

USE_PATH	将php.ini文件中的include_path应用到相对路径上. 内建函数fopen()在指定第三个参数为TRUE时将会设置这个选项.
STREAM_USE_URL	设置这个选项后, 将只能打开远端URL. 对于php://, file://, zlib://, bzip2://这些URL包装器并不认为它们是远端URL.
ENFORCE_SAFE_MODE	尽管这个常量这样命名, 但实际上设置这个选项后仅仅是启用了安全模式(php.ini文件中的safe_mode指令)的强制检查. 如果没有设置这个选项将导致跳过safe_mode的检查(不论INI设置中safe_mode如何设置)
REPORT_ERRORS	在指定的资源打开过程中碰到错误时, 如果设置了这个选项则将产生错误报告.
STREAM_MUST_SEEK	对于某些流, 比如套接字, 是不可以seek的(随机访问); 这类文件句柄, 只有在特定情况下才可以seek. 如果调用作用域指定这个选项, 并且包装器检测到它不能保证可以seek, 将会拒绝打开这个流.
STREAM_WILL_CAST	如果调用作用域要求流可以被转换到stdio或posix文件描述符, 则应该给open_wrapper函数传递这个选项, 以保证在I/O操作发生之前就失败
STREAM_ONLY_GET_HEADERS	标识只需要从流中请求元数据. 实际上这是用于http包装器, 获取http_response_headers全局变量而不真正的抓取远程文件内容.
STREAM_DISABLE_OPEN_BASEDIR	类似safe_mode检查, 不设置这个选项则会检查INI设置open_basedir, 如果指定这个选项则可以绕过这个默认的检查
STREAM_OPEN_PERSISTENT	告知流包装层, 所有内部分配的空间都采用持久化分配, 并将关联的资源注册到持久化列表中.
IGNORE_PATH	如果不指定, 则搜索默认的包含路径. 多数URL包装器都忽略这个选项.
IGNORE_URL	提供这个选项时, 流包装层只打开本地文件. 所有的is_url包装器都将被忽略.

最后的NULL参数是char **类型, 它最初是用来设置匹配路径, 如果path指向普通文件URL, 则去掉file://部分, 保留直接的文件路径用于传统的文件名操作. 这个参数仅仅是以前引擎内部处理使用的.

此外, 还有php_stream_open_wrapper()的一个扩展版本:

```
php_stream *php_stream_open_wrapper_ex(char *path, char *mode, int options, char **opened
`
```

最后一个参数context允许附加的控制, 并可以得到包装器内的通知. 你将在第16章看到这个参数的细节.

传输层包装

尽管传输流和fopen包装流是相同的组件组成的, 但它的注册策略和其他的流不同. 从某种程度上来说, 这是因为用户空间对它们的访问方式的不同造成的, 它们需要实现基于套接字的其他因子.

从扩展开发者角度来看, 打开传输流的过程是相同的. 下面是对fsockopen()的实现:

```
PHP_FUNCTION(sample5_fsockopen)
    php_stream *stream;
    char *host, *transport, *errstr = NULL;
    int host_len, transport_len, implicit_tcp = 1, errcode = 0;
    long port =
    int options = ENFORCE_SAFE_MODE;
    int flags = STREAM_XPORT_CLIENT | STREAM_XPORT_CONNECT;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s|l",
                &host, &host_len, &port) == FAILURE) {
        return;
    }
    if (port) {
        int implicit_tcp = 1;
        if (strstr(host, "://")) {
            /* A protocol was specified,
             * no need to fall back on tcp:// */
            implicit_tcp = 0;
        }
        transport_len = spprintf(&transport, 0, "%s%s:%d",
                implicit_tcp ? "tcp://" : "", host, port);
    } else {
        /* When port isn't specified
         * we can safely assume that a protocol was
         * (e.g. unix:// or udg://) */
        transport = host;
        transport_len = host_len;
    }
    stream = php_stream_xport_create(transport, transport_len,
            options, flags,
            NULL, NULL, NULL, &errstr, &errcode);
    if (transport != host) {
        efree(transport);
    }
    if (errstr) {
        php_error_docref(NULL TSRMLS_CC, E_WARNING, "[%d] %s",
                errcode, errstr);
        efree(errstr);
    }
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

这个函数的基础构造和前面的fopen示例是一样的. 不同在于host和端口号使用不同的参数指定, 接着为了给出一个传输流URL就必须将它们合并到一起. 在产生了一个有意义的路径后, 将它传递给php_stream_xport_create()函数, 方式和fopen()使用的php_stream_open_wrapper()API一样.
php_stream_xport_create()的原型如下:

```
php_stream *php_stream_xport_create(char *xport, int xport_len,
    int options, int flags,
    const char *persistent_id,
    struct timeval *timeout,
    php_stream_context *context,
    char **errstr, int *errcode);
```

每个参数的含义如下:

<table>
<tr><td>xport</td><td>基于URI的传输描述符. 对于基于inet的套接字流, 它可以是tcp://127.0.0.1:80, udp://10.0.0.1:53, ssl://169.254.13.24:445等. 此外, UNIX域传输协议unix:///path/to/socket,udg:///path/to/dgramsocket等都是合法的. xport_len指定了xport的长度, 因此xport是二进制安全的.</td></tr>
<tr><td>options</td><td>这个值是由前面php_stream_open_wrapper()中介绍的选项通过按位或组成的值.</td></tr>
<tr><td>flags</td><td>由STREAM_XPORT_CLIENT或STREAM_XPORT_SERVER之一与下面另外一张表中将列出的STREAM_XPORT_*常量通过按位或组合得到的值.</td></tr>
<tr><td>persistent_id</td><td>如果请求的传输流需要在请求间持久化, 调用作用域可以提供一个key名字描述连接. 指定这个值为NULL创建非持久化连接; 指定为唯一的字符串值将尝试首先从持久化池中查找已有的传输流, 或者没有找到时就创建一个新的持久化流.</td></tr>
<tr><td>timeout</td><td>在超时返回失败之前连接的尝试时间. 如果这个值传递为NULL则使用php.ini中指定的默认超时值. 这个参数对服务端传输流没有意义.</td></tr>
<tr><td>errstr</td><td>如果在选定的套接字上创建, 连接, 绑定或监听时发生错误, 这里传递的char *引用值将被设置为一个描述发生错误原因的字符串. errstr初始应该指向的是NULL; 如果在返回时它被设置了值, 则调用作用域有责任去释放这个字符串相关的内存.</td></tr>
<tr><td>errcode</td><td>通过errstr返回的错误消息对应的数值错误代码.php_stream_xport_create()的flags参数中使用了STREAM_XPORT_*一族常量定义如下:
<table>
<tr><td>STREAM_XPORT_CLIENT</td><td>本地端将通过传输层和远程资源建立连接. 这个标记通常和STREAM_XPORT_CONNECT或STREAM_XPORT_CONNECT_ASYNC联合使用.</td></tr>
<tr><td>STREAM_XPORT_SERVER</td><td>本地端将通过传输层accept连接. 这个标记通常和STREAM_XPORT_BIND以及STREAM_XPORT_LISTEN一起使用.</td></tr>
<tr><td>STREAM_XPORT_CONNECT</td><td>用以说明建立远程资源连接是传输流创建的一部分. 在创建客户端传输流时省略这个标记是合法的, 但是这样做就要求手动的调用php_stream_xport_connect().</td></tr>
<tr><td>STREAM_XPORT_CONNECT_ASYNC</td><td>尝试连接到远程资源, 但不阻塞。</td></tr>
<tr><td>STREAM_XPORT_BIND</td><td>将传输流绑定到本地资源. 用在服务端传输流时,这将使得accept连接的传输流准备端口, 路径或特定的端点标识符等信息.</td></tr>
<tr><td>STREAM_XPORT_LISTEN</td><td>在已绑定的传输流端点上监听到来的连接. 这通常用于基于流的传输协议, 比如: tcp://, ssl://,unix://.</td></tr>
</table>
</td></tr>
</table>

目录访问

fopen包装器支持目录访问, 比如file://和ftp://, 还有第三种流打开函数也可以用于目录访问, 下面是对opendir()的实现:

```
PHP_FUNCTION(sample5_opendir)
{
    php_stream *stream;
    char *path;
    int path_len, options = ENFORCE_SAFE_MODE | REPORT_ERRORS;
    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
        &path, &path_len) == FAILURE) {
        return;
    }
    stream = php_stream_opendir(path, options, NULL);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

同样的, 也可以为某个特定目录打开一个流, 比如本地文件系统的目录名或支持目录访问的URL格式资源. 这里我们又看到了options参数, 它和原来的含义一样, 第三个参数NULL原型是php_stream_context类型.

在目录流打开后, 和文件以及传输流一样, 返回给用户空间.

特殊流

还有一些特殊类型的流不能归类到fopen/transport/directory中. 它们中每一个都有自己独有的API:

```
php_stream *php_stream_fopen_tmpfile(void);
php_stream *php_stream_fopen_temporary_file(const char *dir, const char *pfx, char **open
```

创建一个可seek的缓冲区流用于读写. 在关闭时, 这个流使用的所有临时资源, 包括所有的缓冲区(无论是在内存还是磁盘), 都将被释放. 使用这一组API中的后一个函数, 允许临时文件被以特定的格式命名放到指定路径. 这些内部API调用被用户空间的tmpfile()函数隐藏.

```
php_stream *php_stream_fopen_from_fd(int fd, const char *mode, const char *persistent_id)
php_stream *php_stream_fopen_from_file(FILE *file, const char *mode);
php_stream *php_stream_fopen_from_pipe(FILE *file, const char *mode);
```

这3个API方法接受已经打开的FILE *资源或文件描述符ID, 使用流API的某种操作包装. fd格式的接口不会搜索匹配你前面看到过的fopen函数打开的资源, 但是它会注册持久化的资源, 后续的fopen可以使用到这个持久化资源.

links

- 目录

- 14 流式访问
- 14.2 访问流

静态资源操作

一个基于流的原子操作并不需要实际的实例. 下面这些API仅仅使用URL执行这样的操作:

```
int php_stream_stat_path(char *path, php_stream_statbuf *ssb);
```

和前面的php_stream_stat()类似, 这个函数提供了一个对POSIX的stat()函数协议依赖的包装. 要注意, 并不是所有的协议都支持URL记法, 并且即便支持也可能不能报告出statbuf结构体中的所有成员值. 一定要检查php_stream_stat_path()失败时的返回值, 0标识成功, 要知道, 不支持的元素返回时其值将是默认的0.

```
int php_stream_stat_path_ex(char *path, int flags,
    php_stream_statbuf *ssb, php_stream_context *context);
```

这个php*stream_url_stat()*的扩展版本允许传递另外两个参数. 第一个是*flags,* 它的值可以是下面的*PHP_STERAM_URL_STAT**(下面省略PHP*STREAM_URL_STAT*前缀)一族常量的按位或的结果. 还有一个是context参数, 它在其他的一些流函数中也有出现,我们将在第16章去详细学习.

LINK	原始的php_stream_stat_path()对于符号链接或目录将会进行解析直到碰到协议定义的结束资源. 传递PHP_STREAM_URL_STAT_LINK标记将导致php_stream_stat_path()返回请求资源的信息而不会进行符号链接的解析.(译注: 我们可以这样理解, 没有这个标记, 底层使用stat(), 如果有这个标记,底层使用lstat(), 关于stat()和lstat()的区别, 请查看*nix手册)
QUIET	默认情况下, 如果在执行URL的stat操作过程中碰到错误, 包括文件未找到错误, 都将通过php的错误处理机制触发. 传递QUIET标记可以使得php_stream_stat_path()返回而不报告错误.

```
int php_stream_mkdir(char *path, int mode, int options,
    php_stream_context *context);
int php_stream_rmdir(char *path, int options,
    php_stream_context *context);
```

创建和删除目录也会如你期望的工作. 这里的options参数和前面的php_stream_open_wrapper()函数的同名参数含义一致. 对于php_stream_mkdir(), 还有一个参数mode用于指定一个八进制的值表明读写执行权限.

links

- 目录
- 14.2 访问流
- 14.4 小结

小结

本章中你接触了一些基于流的I/O的内部表象. 下一章将演示做呢样实现自己的协议包装, 甚至是定义自己的流类型.

links

* 目录
* 14.3 访问流
* 15 实现流

15 流的实现

目录

php的流最强力的特性之一是它可以访问众多数据源: 普通文件, 压缩文件, 网络透明 通道, 加密网络, 命名管道以及域套接字, 它们对于用户空间以及内部都是统一的API.

links

- 目录
- 14.5 小结
- 15.1 php流的表象之下

15.1 php流的表象之下

对于给定的流实例, 比如文件流和网络流, 它们的不同在于上一一章你使用的流创建函数返回的php_stream结构体中的ops成员.

```
typedef struct _php_stream {
    ...
    php_stream_ops *ops;
    ...
} php_stream;
```

php_stream_ops结构体定义的是一个函数指针集合以及一个描述标记.

```
typedef struct _php_stream_ops {
    size_t (*write)(php_stream *stream, const char *buf,
                            size_t count TSRMLS_DC);
    size_t (*read)(php_stream *stream, char *buf,
                            size_t count TSRMLS_DC);
    int    (*close)(php_stream *stream, int close_handle
                            TSRMLS_DC);
    int    (*flush)(php_stream *stream TSRMLS_DC);

    const char *label;

    int (*seek)(php_stream *stream, off_t offset, int whence,
                            off_t *newoffset TSRMLS_DC);
    int (*cast)(php_stream *stream, int castas, void **ret
                            TSRMLS_DC);
    int (*stat)(php_stream *stream, php_stream_statbuf *ssb
                            TSRMLS_DC);
    int (*set_option)(php_stream *stream, int option,int value,
                            void *ptrparam TSRMLS_DC);
} php_stream_ops;
```

当流访问函数比如php_stream_read()被调用时, 流包装层实际上解析调用了stream->ops中对应的函数, 这样实际调用的就是当前流类型特有的read实现. 比如, 普通文件的流ops结构体中的read函数实现如下(实际的该实现比下面的示例复杂一点):

```
size_t php_stdio_read(php_stream *stream, char *buf,
                                size_t count TSRMLS_DC)
{
    php_stdio_stream_data *data =
                (php_stdio_stream_data*)stream->abstract;
    return read(data->fd, buf, count);
}
```

而compress.zlib流使用的ops结构体中则read则指向的是如下的函数:

```
size_t php_zlib_read(php_stream *stream, char *buf,
                                size_t count TSRMLS_DC)
{
    struct php_gz_stream_data_t *data =
            (struct php_gz_stream_data_t *) stream->abstract;

    return gzread(data->gz_file, buf, count);
}
```

这里第一点需要注意的是ops结构体指向的函数指针常常是对数据源真正的读取函数的一个瘦代理. 在上面两个例子中, 标准I/O流使用posix的read()函数, 而zlib流使用的是libz的gzread()函数.

你可能还注意到了, 这里使用了stream->abstract元素. 这是流实现的一个便利指针, 它可以被用于获取各种相关的捆绑信息. 在上面的例子中, 指向自定义结构体的指针, 用于存储底层read函数要使用的文件描述符.

还有一件你可能注意到的事情是php_stream_ops结构体中的每个函数都期望一个已有的流实例, 但是怎样得到实例呢? abstract成员是怎样设置的以及什么时候流指示使用哪个ops结构体? 答案就在你在上一章使用过的第一个打开流的函数(php_stream_open_wrapper())中.

当这个函数被调用时, php的流包装层尝试基于传递的URL中的scheme://部分确定请求的是什么协议. 这样它就可以在已注册的php包装器中查找对应的php_stream_wrapper项. 每个php_stream_wrapper结构体都可以取到自己的ops元素, 它指向一个php_stream_wrapper_ops结构体:

```
typedef struct _php_stream_wrapper_ops {
    php_stream *(*stream_opener)(php_stream_wrapper *wrapper,
                        char *filename, char *mode,
                        int options, char **opened_path,
                        php_stream_context *context
                        STREAMS_DC TSRMLS_DC);
    int (*stream_closer)(php_stream_wrapper *wrapper,
                        php_stream *stream TSRMLS_DC);
    int (*stream_stat)(php_stream_wrapper *wrapper,
                        php_stream *stream,
                        php_stream_statbuf *ssb
                        TSRMLS_DC);
    int (*url_stat)(php_stream_wrapper *wrapper,
                        char *url, int flags,
                        php_stream_statbuf *ssb,
                        php_stream_context *context
                        TSRMLS_DC);
    php_stream *(*dir_opener)(php_stream_wrapper *wrapper,
                        char *filename, char *mode,
                        int options, char **opened_path,
                        php_stream_context *context
                        STREAMS_DC TSRMLS_DC);

    const char *label;

    int (*unlink)(php_stream_wrapper *wrapper, char *url,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);

    int (*rename)(php_stream_wrapper *wrapper,
```

```
                        char *url_from, char *url_to,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);


    int (*stream_mkdir)(php_stream_wrapper *wrapper,
                        char *url, int mode, int options,
                        php_stream_context *context
                        TSRMLS_DC);
    int (*stream_rmdir)(php_stream_wrapper *wrapper, char *url,
                        int options,
                        php_stream_context *context
                        TSRMLS_DC);
} php_stream_wrapper_ops;
```

这里, 流包装层调用wrapper->ops->stream_opener(), 它将执行包装器特有的操作创建流实例, 赋值恰当的php_stream_ops结构体, 绑定相关的抽象数据.

dir_opener()函数和stream_opener()提供相同的基础服务; 不过, 它是对php_stream_opendir()这个API调用的响应, 并且通常会绑定一个不同的php_stream_ops结构体到返回的实例. stat()和close()函数在这一层上是重复的, 这样做是为了给包装器的这些操作增加协议特有的逻辑.

其他的函数则允许执行静态流操作而不用实际的创建流实例. 回顾这些流API调用, 它们并不实际返回php_stream对象, 你马上就会看到它们的细节.

> 尽管在php 4.3中引入流包装层时, url_stat在内部作为一个包装器的ops函数存在, 但直到php 5.0它才开始被使用. 此外, 最后的3个函数, rename(), stream_mkdir()以及stream_rmdir()一直到php 5.0才引入, 在这个版本之前, 它们并不在包装器的ops结构中.

links

- 目录
- 15 流的实现
- 15.2 包装器操作

包装器操作

除了url_stat()函数, 包装器操作中在const char *label元素之前的每个操作都可以用于激活的流实例上. 每个函数的意义如下:

stream_opener()	实例化一个流实例. 当某个用户空间的fopen()函数被调用时, 这个函数将被调用. 这个函数返回的php_stream实例是fopen()函数返回的文件资源句柄的内部表示. 集成函数比如file(), file_get_contents(), file_put_contents(), readfile()等等, 在请求包装资源时, 都使用这个包装器ops.
stream_closer()	当一个流实例结束其生命周期时这个函数被调用. stream_opener()时分配的所有资源都应该在这个函数中被释放.
stream_stat()	类似于用户空间的fstat()函数, 这个函数应该填充ssb结构体(实际上只包含一个struct statbuf sb结构体成员),
dir_opener()	和stream_opener()行为一致, 不过它是调用opendir()一族的用户空间函数时被调用的. 目录流使用的底层流实现和文件流遵循相同的规则;不过目录流只需要返回包含在打开的目录中找到的文件名的记录, 它的大小为struct dirent这个结构体的大小.

静态包装器操作

包装器操作函数中的其他函数是在URI路径上执行原子操作, 具体取决于包装器协议. 在php4.3的php_stream_wrapper_ops结构体中只有url_stat()和unlink(); 其他的方式是到php 5.0后才定义的, 编码时应该适时的使用#ifdef块说明.

url_stat()	stat()族函数使用, 返回文件元数据, 比如访问授权, 大小, 类型; 以及访问, 修改,创建时间. 尽管这个函数是在php 4.3引入流包装层时出现在php_stream_wrapper_ops结构体中的, 但直到php 5.0才被用户空间的stat()函数使用.
unlink()	和posix文件系统的同名函数语义相同, 它执行文件删除. 如果对于当前的包装器删除没有意义, 比如内建的http://包装器, 这个函数应该被定义为NULL, 以便内核去引发适当的错误消息.
rename()	当用户空间的rename()函数的参数$from和$to参数指向的是相同的底层包装器实现, php则将这个重命名请求分发到包装器的rename函数.
mkdir() & rmdir()	这两个函数直接映射到对应的用户空间函数.

links

* 目录
* 15.1 php流的表象之下
* 15.3 实现一个包装器

实现一个包装器

为了演示包装器和流操作的内部工作原理, 我们需要重新实现php手册的stream_wrapper_register()一页示例中的var://包装器.

此刻, 首先从下面功能完整的变量流包装实现开始. 构建他, 并开始检查每一块的工作原理.

> 译注: 为了方便大家阅读, 对代码的注释进行了适量补充调整, 此外, 由于phpapi的调整, 原著中的代码不能直接在译者使用的php-5.4.10中运行, 进行了适当的修改. 因此下面代码结构可能和原著略有不同, 请参考阅读.(下面opendir的例子也进行了相应的修改)

config.m4

```
PHP_ARG_ENABLE(varstream,whether to enable varstream support,
[  enable-varstream      Enable varstream support])

if test "$PHP_VARSTREAM" = "yes"; then
  AC_DEFINE(HAVE_VARSTREAM,1,[Whether you want varstream])
  PHP_NEW_EXTENSION(varstream, varstream.c, $ext_shared)
fi
```

php_varstream.h

```
#ifndef PHP_VARSTREAM_H
#define PHP_VARSTREAM_H

extern zend_module_entry varstream_module_entry;
#define phpext_varstream_ptr &varstream_module_entry

#ifdef PHP_WIN32
#   define PHP_VARSTREAM_API __declspec(dllexport)
#elif defined(__GNUC__) && __GNUC__ >= 4
#   define PHP_VARSTREAM_API __attribute__ ((visibility("default")))
#else
#   define PHP_VARSTREAM_API
#endif

#ifdef ZTS
#include "TSRM.h"
#endif

PHP_MINIT_FUNCTION(varstream);
PHP_MSHUTDOWN_FUNCTION(varstream);

#define PHP_VARSTREAM_WRAPPER       "var"
#define PHP_VARSTREAM_STREAMTYPE    "varstream"

/* 变量流的抽象数据结构 */
typedef struct _php_varstream_data {
    off_t   position;
```

```
    char    *varname;
    int     varname_len;
} php_varstream_data;

#ifdef ZTS
#define VARSTREAM_G(v) TSRMG(varstream_globals_id, zend_varstream_globals *, v)
#else
#define VARSTREAM_G(v) (varstream_globals.v)
#endif

#endif
```

varstream.c

```
#ifdef HAVE_CONFIG_H
#include "config.h"
#endif

#include "php.h"
#include "php_ini.h"
#include "ext/standard/info.h"
#include "ext/standard/url.h"
#include "php_varstream.h"

static size_t php_varstream_write(php_stream *stream,
                const char *buf, size_t count TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var;
    size_t newlen;

    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), data->varname,
            data->varname_len + 1,(void**)&var) == FAILURE) {
        /* 变量不存在, 直接创建一个字符串类型的变量, 并保存新传递进来的内容 */
       zval *newval;
       MAKE_STD_ZVAL(newval);
       ZVAL_STRINGL(newval, buf, count, 1);
       /* 将新的zval *放到变量中 */
       zend_hash_add(&EG(symbol_table), data->varname,
           data->varname_len + 1, (void*)&newval,
           sizeof(zval*), NULL);
       return count;
    }
    /* 如果需要, 让变量可写. 这里实际上处理的是写时复制 */
    SEPARATE_ZVAL_IF_NOT_REF(var);
    /* 转换为字符串类型 */
    convert_to_string_ex(var);
    /* 重置偏移量(译注: 相比于正常的文件系统, 这里的处理实际上不支持文件末尾的空洞创建, 读者如果熟悉*ni
    if (data->position > Z_STRLEN_PP(var)) {
        data->position = Z_STRLEN_PP(var);
    }
    /* 计算新的字符串长度 */
    newlen = data->position + count;
    if (newlen < Z_STRLEN_PP(var)) {
        /* 总长度不变 */
```

```
        newlen = Z_STRLEN_PP(var);
    } else if (newlen > Z_STRLEN_PP(var)) {
        /* 重新调整缓冲区大小以保存新内容 */
        Z_STRVAL_PP(var) =erealloc(Z_STRVAL_PP(var),newlen+1);
        /* 更新字符串长度 */
        Z_STRLEN_PP(var) = newlen;
        /* 确保字符串NULL终止 */
        Z_STRVAL_PP(var)[newlen] = 0;
    }
    /* 将数据写入到变量中 */
    memcpy(Z_STRVAL_PP(var) + data->position, buf, count);
    data->position += count;

    return count;
}

static size_t php_varstream_read(php_stream *stream,
                char *buf, size_t count TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var, copyval;
    int got_copied = 0;
    size_t toread = count;

    if (zend_hash_find(&EG(symbol_table), data->varname,
        data->varname_len + 1, (void**)&var) == FAILURE) {
        /* 变量不存在, 读不到数据, 返回0字节长度 */
        return 0;
    }
    copyval = **var;
    if (Z_TYPE(copyval) != IS_STRING) {
        /* 对于非字符串类型变量, 创建一个副本进行读, 这样对于只读的变量, 就不会改变其原始类型 */
        zval_copy_ctor(&copyval);
        INIT_PZVAL(&copyval);
        got_copied = 1;
    }
    if (data->position > Z_STRLEN(copyval)) {
        data->position = Z_STRLEN(copyval);
    }
    if ((Z_STRLEN(copyval) - data->position) < toread) {
        /* 防止读取到变量可用缓冲区外的内容 */
        toread = Z_STRLEN(copyval) - data->position;
    }
    /* 设置缓冲区 */
    memcpy(buf, Z_STRVAL(copyval) + data->position, toread);
    data->position += toread;

    /* 如果创建了副本, 则释放副本 */
    if (got_copied) {
        zval_dtor(&copyval);
    }

    /* 返回设置到缓冲区的字节数 */
    return toread;
}

static int php_varstream_closer(php_stream *stream,
                            int close_handle TSRMLS_DC)
{
```

```
    php_varstream_data *data = stream->abstract;

    /* 释放内部结构避免泄露 */
    efree(data->varname);
    efree(data);

    return 0;
}

static int php_varstream_flush(php_stream *stream TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;
    zval **var;

    /* 根据不同情况, 重置偏移量 */
    if (zend_hash_find(&EG(symbol_table), data->varname,
                    data->varname_len + 1, (void**)&var)
                    == SUCCESS) {
        if (Z_TYPE_PP(var) == IS_STRING) {
            data->position = Z_STRLEN_PP(var);
        } else {
            zval copyval = **var;
            zval_copy_ctor(&copyval);
            convert_to_string(&copyval);
            data->position = Z_STRLEN(copyval);
            zval_dtor(&copyval);
        }
    } else {
        data->position = 0;
    }

    return 0;
}

static int php_varstream_seek(php_stream *stream, off_t offset,
                    int whence, off_t *newoffset TSRMLS_DC)
{
    php_varstream_data *data = stream->abstract;

    switch (whence) {
        case SEEK_SET:
            data->position = offset;
            break;
        case SEEK_CUR:
            data->position += offset;
            break;
        case SEEK_END:
        {
            zval **var;
           size_t curlen = 0;

           if (zend_hash_find(&EG(symbol_table),
                   data->varname,    data->varname_len + 1,
                   (void**)&var) == SUCCESS) {
              if (Z_TYPE_PP(var) == IS_STRING) {
                  curlen = Z_STRLEN_PP(var);
              } else {
                  zval copyval = **var;
                  zval_copy_ctor(&copyval);
```

```
                    convert_to_string(&copyval);
                    curlen = Z_STRLEN(copyval);
                    zval_dtor(&copyval);
                }
            }

            data->position = curlen + offset;
            break;
        }
    }

    /* 防止随机访问指针移动到缓冲区开始位置之前 */
    if (data->position < 0) {
        data->position = 0;
    }

    if (newoffset) {
        *newoffset = data->position;
    }

    return 0;
}

static php_stream_ops php_varstream_ops = {
    php_varstream_write,
    php_varstream_read,
    php_varstream_closer,
    php_varstream_flush,
    PHP_VARSTREAM_STREAMTYPE,
    php_varstream_seek,
    NULL, /* cast */
    NULL, /* stat */
    NULL, /* set_option */
};

/* Define the wrapper operations */
static php_stream *php_varstream_opener(
            php_stream_wrapper *wrapper,
            char *filename, char *mode, int options,
            char **opened_path, php_stream_context *context
            STREAMS_DC TSRMLS_DC)
{
    php_varstream_data *data;
    php_url *url;

    if (options & STREAM_OPEN_PERSISTENT) {
        /* 按照变量流的定义, 是不能持久化的
         * 因为变量在请求结束后将被释放
         */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unable to open %s persistently",
                                        filename);
        return NULL;
    }

    /* 标准URL解析: scheme://user:pass@host:port/path?query#fragment */
    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
```

```
            TSRMLS_CC, "Unexpected error parsing URL");
        return NULL;
    }
    /* 检查是否有变量流URL必须的元素host, 以及scheme是否是var */
    if (!url->host || (url->host[0] == 0) ||
        strcasecmp("var", url->scheme) != 0) {
        /* Bad URL or wrong wrapper */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Invalid URL, must be in the form: "
                     "var://variablename");
        php_url_free(url);
        return NULL;
    }

    /* 创建一个数据结构保存协议信息(变量流协议重要是变量名, 变量名长度, 当前偏移量) */
    data = emalloc(sizeof(php_varstream_data));
    data->position = 0;
    data->varname_len = strlen(url->host);
    data->varname = estrndup(url->host, data->varname_len + 1);
    /* 释放前面解析出来的url占用的内存 */
    php_url_free(url);

    /* 实例化一个流, 为其赋予恰当的流ops, 绑定抽象数据 */
    return php_stream_alloc(&php_varstream_ops, data, 0, mode);
}

static php_stream_wrapper_ops php_varstream_wrapper_ops = {
    php_varstream_opener, /* 调用php_stream_open_wrapper(sprintf("%s://xxx", PHP_VARSTREAM
    NULL, /* stream_close */
    NULL, /* stream_stat */
    NULL, /* url_stat */
    NULL, /* dir_opener */
    PHP_VARSTREAM_WRAPPER,
    NULL, /* unlink */
#if PHP_MAJOR_VERSION >= 5
    /* PHP >= 5.0 only */
    NULL, /* rename */
    NULL, /* mkdir */
    NULL, /* rmdir */
#endif
};

static php_stream_wrapper php_varstream_wrapper = {
    &php_varstream_wrapper_ops,
    NULL, /* abstract */
    0, /* is_url */
};

PHP_MINIT_FUNCTION(varstream)
{
    /* 注册流包装器:
     * 1. 检查流包装器名字是否正确(符合这个正则: /^[a-zA-Z0-9+.-]+$/)
     * 2. 将传入的php_varstream_wrapper增加到url_stream_wrappers_hash这个HashTable中, key为PH
     */
    if (php_register_url_stream_wrapper(PHP_VARSTREAM_WRAPPER,
            &php_varstream_wrapper TSRMLS_CC)==FAILURE) {
        return FAILURE;
    }
    return SUCCESS;
```

```
}

PHP_MSHUTDOWN_FUNCTION(varstream)
{
    /* 卸载流包装器: 从url_stream_wrappers_hash中删除 */
    if (php_unregister_url_stream_wrapper(PHP_VARSTREAM_WRAPPER
                                TSRMLS_CC) == FAILURE) {
        return FAILURE;
    }
    return SUCCESS;
}

zend_module_entry varstream_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    "varstream",
    NULL,
    PHP_MINIT(varstream),
    PHP_MSHUTDOWN(varstream),
    NULL,
    NULL,
    NULL,
#if ZEND_MODULE_API_NO >= 20010901
    "0.1",
#endif
    STANDARD_MODULE_PROPERTIES
};

#ifdef COMPILE_DL_VARSTREAM
ZEND_GET_MODULE(varstream)
#endif
```

在构建加载扩展后, php就可以处理以var://开始的URL的请求, 它的行为和手册中用户空间实现的行为一致.

内部实现

首先你注意到的可能是这个扩展完全没有暴露用户空间函数. 它所做的只是在MINIT函数中调用了一个核心PHPAPI的钩子, 将var协议和我们定义的包装器关联起来:

```
static php_stream_wrapper php_varstream_wrapper = {
    &php_varstream_wrapper_ops,
    NULL, /* abstract */
    0, /* is_url */
}
```

很明显, 最重要的元素就是ops, 它提供了访问特定流包装器的创建以及检查函数. 你可以安全的忽略abstract属性, 它仅在运行时使用, 在初始化定义时, 它只是作为一个占位符. 第三个元素is_url, 它告诉php在使用这个包装器时是否考虑php.ini中的allow_url_fopen选项. 如果这个值非0, 并且将allow_url_fopen设置为false, 则这个包装器不能被脚本使用.

在本章前面你已经知道, 调用用户空间函数比如fopen将通过这个包装器的ops元素得到

php_varstream_wrapper_ops, 这样去调用流的打开函数php_varstream_opener.

这个函数的第一块代码检查是否请求持久化的流:

```
if (options & STREAM_OPEN_PERSISTENT) {
```

对于很多包装器这样的请求是合法的. 然而目前的情况这个行为没有意义. 一方面用户空间变量的定义就是临时的, 另一方面, varstream的实例化代价很低, 这就使得持久化的优势很小.

像流包装层报告错误很简单, 只需要返回一个NULL值而不是流实例即可. 流包装层透出到用户空间的失败消息并不会说明具体的错误, 只是说明不能打开URL. 要想给开发者暴露更多的错误信息, 可以在返回之前使用php_stream_wrapper_log_error()函数.

```
php_stream_wrapper_log_error(wrapper, options
    TSRMLS_CC, "Unable to open %s persistently",
                                filename);
return NULL;
```

URL解析

实例化varstream的下一步需要一个人类可读的URL, 将它分块放入到一个易管理的结构体中. 幸运的是它使用了和用户空间url_parse()函数相同的机制. 如果URL成功解析, 将会分配一个php_url结构体并设置合适的值. 如果在URL中没有某些值, 在返回的php_url中对应的将被设置为NULL. 这个结构体必须在离开php_varstream_opener函数之前被显式释放, 否则它的内存将会泄露:

```
typedef struct php_url {
    /* scheme://user:pass@host:port/path?query#fragment */
    char *scheme;
    char *user;
    char *pass;
    char *host;
    unsigned short port;
    char *path;
    char *query;
    char *fragment;
} php_url;
```

最后, varstream包装器创建了一个数据结构, 保存了流指向的变量名, 读取时的当前位置. 这个结构体将在流的读取和写入函数中用于获取变量, 并且将在流结束使用时由php_varstream_close函数释放.

opendir()

读写变量内容的实现可以再次进行扩展. 这里可以加入一个新的特性, 允许使用目录函数读取数组中的key. 在你的php_varstream_wrapper_ops结构体之前增加下面的代码:

```
static size_t php_varstream_readdir(php_stream *stream,
```

```
                char *buf, size_t count TSRMLS_DC)
{
    php_stream_dirent *ent = (php_stream_dirent*)buf;
    php_varstream_dirdata *data = stream->abstract;
    char *key;
    int type, key_len;
    long idx;

    /* 查找数组中的key */
    type = zend_hash_get_current_key_ex(Z_ARRVAL_P(data->arr),
                    &key, &key_len, &idx, 0, &(data->pos));

    /* 字符串key */
    if (type == HASH_KEY_IS_STRING) {
        if (key_len >= sizeof(ent->d_name)) {
            /* truncate long keys to maximum length */
            key_len = sizeof(ent->d_name) - 1;
        }
        /* 设置到目录结构上 */
        memcpy(ent->d_name, key, key_len);
        ent->d_name[key_len] = 0;
    /* 数值key */
    } else if (type == HASH_KEY_IS_LONG) {
        /* 设置到目录结构上 */
        snprintf(ent->d_name, sizeof(ent->d_name), "%ld",idx);
    } else {
        /* 迭代结束 */
        return 0;
    }
    /* 移动数组指针(位置记录到流的抽象结构中) */
    zend_hash_move_forward_ex(Z_ARRVAL_P(data->arr),
                                    &data->pos);
    return sizeof(php_stream_dirent);
}

static int php_varstream_closedir(php_stream *stream,
                            int close_handle TSRMLS_DC)
{
    php_varstream_dirdata *data = stream->abstract;

    zval_ptr_dtor(&(data->arr));
    efree(data);
    return 0;
}

static int php_varstream_dirseek(php_stream *stream,
                    off_t offset, int whence,
                    off_t *newoffset TSRMLS_DC)
{
    php_varstream_dirdata *data = stream->abstract;

    if (whence == SEEK_SET && offset == 0) {
        /* 重置数组指针 */
        zend_hash_internal_pointer_reset_ex(
                    Z_ARRVAL_P(data->arr), &(data->pos));
        if (newoffset) {
            *newoffset = 0;
        }
        return 0;
```

```
    }
    /* 不支持其他类型的随机访问 */
    return -1;
}

static php_stream_ops php_varstream_dirops = {
    NULL, /* write */
    php_varstream_readdir,
    php_varstream_closedir,
    NULL, /* flush */
    PHP_VARSTREAM_DIRSTREAMTYPE,
    php_varstream_dirseek,
    NULL, /* cast */
    NULL, /* stat */
    NULL, /* set_option */
};

static php_stream *php_varstream_opendir(
            php_stream_wrapper *wrapper,
            char *filename, char *mode, int options,
            char **opened_path, php_stream_context *context
            STREAMS_DC TSRMLS_DC)
{
    php_varstream_dirdata *data;
    php_url *url;
    zval **var;

    /* 不支持持久化流 */
    if (options & STREAM_OPEN_PERSISTENT) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Unable to open %s persistently",
                filename);
        return NULL;
    }

    /* 解析URL */
    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Unexpected error parsing URL");
        return NULL;
    }
    /* 检查请求URL的正确性 */
    if (!url->host || (url->host[0] == 0) ||
            strcasecmp("var", url->scheme) != 0) {
        /* Bad URL or wrong wrapper */
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Invalid URL, must be in the form: "
                "var://variablename");
        php_url_free(url);
        return NULL;
    }

    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), url->host,
                strlen(url->host) + 1, (void**)&var) == FAILURE) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "Variable $%s not found", url->host);
        php_url_free(url);
```

```
        return NULL;
    }

    /* 检查变量类型 */
    if (Z_TYPE_PP(var) != IS_ARRAY) {
        php_stream_wrapper_log_error(wrapper, options
                TSRMLS_CC, "$%s is not an array", url->host);
        php_url_free(url);
        return NULL;
    }
    /* 释放前面分配的URL结构 */
    php_url_free(url);

    /* 分配抽象数据结构 */
    data = emalloc(sizeof(php_varstream_dirdata));
    if ( Z_ISREF_PP(var) && Z_REFCOUNT_PP(var) > 1) {
        /* 全拷贝 */
        MAKE_STD_ZVAL(data->arr);
        *(data->arr) = **var;
        zval_copy_ctor(data->arr);
        INIT_PZVAL(data->arr);
    } else {
        /* 写时拷贝 */
        data->arr = *var;
        Z_SET_REFCOUNT_P(data->arr, Z_REFCOUNT_P(data->arr) + 1);
    }
    /* 重置数组指针 */
    zend_hash_internal_pointer_reset_ex(Z_ARRVAL_P(data->arr),
            &data->pos);
    return php_stream_alloc(&php_varstream_dirops,data,0,mode);
}
```

现在, 将你的php_varstream_wrapper_ops结构体中的dir_opener的NULL替换成你的php_varstream_opendir函数. 最后, 将下面新定义的类型放入到你的php_varstream.h文件的php_varstream_data定义下面:

```
#define PHP_VARSTREAM_DIRSTREAMTYPE    "varstream directory"
typedef struct _php_varstream_dirdata {
    zval *arr;
    HashPosition pos;
} php_varstream_dirdata;
```

在你基于fopen()实现的varstream包装器中, 你直接使用持久变量名, 每次执行读写操作时从符号表中获取变量. 而这里, opendir()的实现中获取变量时处理了变量不存在或者类型错误的异常. 你还有一个数组变量的拷贝, 这就说明原数组的改变并不会影响后续的readdir()调用的结果. 原来存储变量名的方式也可以正常工作, 这里只是给出另外一种选择作为演示示例.

由于目录访问是基于成块的目录条目, 而不是字符, 因此这里需要一套独立的流操作. 这个版本中, write没有意义, 因此保持它为NULL. read的实现使用zend_hash_get_current_key_ex()函数将数组映射到目录名. 而随机访问也只是对SEEK_SET有效, 用来响应rewinddir()跳转到数组开始位置.

> 实际上, 目录流并没有使用SEEK_CUR, SEEK_END, 或者除了0之外的偏移量. 在实现目录流操作时, 最好还是涉及你的函数能以某种方式处理这些情况, 以使得在流包装层变化时能够适应其目录随机访

> 问.

links

操纵

5个静态包装器操作中的4个用来处理不是基于I/O的流资源操作. 你已经看到过它们并了解它们的原型; 现在我们看看varstream包装器框架中它们的实现:

unlink

在你的wrapper_ops结构体中增加下面的函数, 它可以让unlink()通过varstream包装器, 拥有和unset()一样的行为:

```
static int php_varstream_unlink(php_stream_wrapper *wrapper,
                        char *filename, int options,
                        php_stream_context *context
                        TSRMLS_DC)
{
    php_url *url;

    url = php_url_parse(filename);
    if (!url) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing URL");
        return -1;
    }
    if (!url->host || (url->host[0] == 0) ||
        strcasecmp("var", url->scheme) != 0) {
        /* URL不合法 */
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Invalid URL, must be in the form: "
                     "var://variablename");
        php_url_free(url);
        return -1;
    }

    /* 从符号表删除变量 */
    //zend_hash_del(&EG(symbol_table), url->host, strlen(url->host) + 1);
    zend_delete_global_variable(url->host, strlen(url->host) + 1 TSRMLS_CC);

    php_url_free(url);
    return 0;
}
```

这个函数的编码量和php_varstream_opener差不多. 唯一的不同在于这里你需要传递变量名给zend_hash_del()去删除变量.

> 译注: 译者的php-5.4.10环境中, 使用unlink()删除变量后, 在用户空间再次读取该变量名的值会导致core dump. 因此上面代码中译者进行了修正, 删除变量时使用了zend_delete_global_variable(), 请读者参考阅读zend_delete_global_variable()函数源代码, 考虑为什么直接用zend_hash_del()删除, 会导致core dump. 下面是译者测试用的用户空间代码:

```
<?php
```

```
$fp = fopen('var://hello', 'r');
fwrite($fp, 'world');
var_dump($hello);
unlink('var://hello');
$a  = $hello;
```

这个函数的代码量应该和php_varstream_opener差不多. 唯一的不同是这里是传递变量名给zend_hash_del()去删除变量.

rename, mkdir, rmdir

为了一致性, 下面给出rename, mkdir, rmdir函数的实现:

```
static int php_varstream_rename(php_stream_wrapper *wrapper,
        char *url_from, char *url_to, int options,
        php_stream_context *context TSRMLS_DC)
{
    php_url *from, *to;
    zval **var;

    /* 来源URL解析 */
    from = php_url_parse(url_from);
    if (!from) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing source");
        return -1;
    }
    /* 查找变量 */
    if (zend_hash_find(&EG(symbol_table), from->host,
                strlen(from->host) + 1,
                (void**)&var) == FAILURE) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "$%s does not exist", from->host);
        php_url_free(from);
        return -1;
    }
    /* 目标URL解析 */
    to = php_url_parse(url_to);
    if (!to) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "Unexpected error parsing dest");
        php_url_free(from);
        return -1;
    }
    /* 变量的改名 */
    Z_SET_REFCOUNT_PP(var, Z_REFCOUNT_PP(var) + 1);
    zend_hash_update(&EG(symbol_table), to->host,
                strlen(to->host) + 1, (void*)var,
                sizeof(zval*), NULL);
    zend_hash_del(&EG(symbol_table), from->host,
                strlen(from->host) + 1);
    php_url_free(from);
    php_url_free(to);
    return 0;
}
```

```
static int php_varstream_mkdir(php_stream_wrapper *wrapper,
                char *url_from, int mode, int options,
                php_stream_context *context TSRMLS_DC)
{
    php_url *url;

    /* URL解析 */
    url = php_url_parse(url_from);
    if (!url) {
       php_stream_wrapper_log_error(wrapper, options
           TSRMLS_CC, "Unexpected error parsing URL");
       return -1;
    }

    /* 检查变量是否存在 */
    if (zend_hash_exists(&EG(symbol_table), url->host,
                    strlen(url->host) + 1)) {
        php_stream_wrapper_log_error(wrapper, options
            TSRMLS_CC, "$%s already exists", url->host);
        php_url_free(url);
        return -1;
    }
    /* EG(uninitialized_zval_ptr)通常是IS_NULL的zval *, 引用计数无限大 */
    zend_hash_add(&EG(symbol_table), url->host,
            strlen(url->host) + 1,
            (void*)&EG(uninitialized_zval_ptr),
            sizeof(zval*), NULL);
    php_url_free(url);
    return 0;
}

static int php_varstream_rmdir(php_stream_wrapper *wrapper,
                char *url, int options,
                php_stream_context *context TSRMLS_DC)
{
    /* 行为等价于unlink() */
    wrapper->wops->unlink(wrapper, url, options,
                                context TSRMLS_CC);
}
```

links

- 目录
- 15.3 实现一个包装器
- 15.5 检查

检查

并不是所有的流操作都涉及到资源的操纵. 有时候也需要查看活动的流在某个时刻的状态, 或检查潜在可打开的资源的状态.

这一节流和包装器的ops函数都是在相同的数据结构php_stream_statbuf上工作的, 它只有一个元素: posix标准的struct statbuf. 当本节的某个函数被调用时, 将尝试填充尽可能多的statbuf元素的成员.

stat

如果设置, 当请求激活流实例的信息时, 将会调用wrapper->ops->stream_stat(). 如果没有设置, 则对应的stream->ops->stat()将会被调用. 无论哪个函数被调用, 都应该尽可能多的向返回的statbuf结构体ssb->sb中填充尽可能多流实例的有用信息. 在普通文件I/O的用法中, 它对应fstat()的标准I/O调用.

url_stat

在流实例外部调用wrapper->ops->url*stat()*取到流资源的元数据. 通常来说, 符号链接和重定向都应该被解析, 直到找到一个真正的资源, 对其通过*stat()*系统调用这样的机制读取统计信息. *url_stat*的*flags*参数允许是下面*PHP_STREAM_URL_STAT**系列的常量值(省略PHP*STREAM_URL_STAT*前缀):

LINK	不解析符号链接和重定向. 而是报告它碰到的第一个节点的信息, 无论是连接还是真正的资源.
QUIET	不报告错误. 注意, 这和许多其他流函数中的REPORT_ERRORS逻辑恰恰相反.

links

- 目录
- 15.4 操纵
- 15.6 小结

小结

无论是暴露远程网络I/O还是本地数据源的流资源, 都允许你的扩展在核心数据上挂在操纵函数的钩子, 避免重新实现单调的描述符管理和I/O缓冲区工作. 这使得它在用户空间环境中更加有用, 更加强大.

下一章将通过对过滤器和上下文的学习结束流包装层的学习, 过滤器和上下文可以用于选择默认的流行为, 甚至过程中修改数据.

links

- 目录
- 15.5 检查
- 16 有趣的流

16 有趣的流

目录

php常被提起的一个特性是流上下文. 这个可选的参数甚至在用户空间大多数流创建相关的函数中都可用, 它作为一个泛化的框架用于向给定包装器或流实现传入/传出额外的信息.

links

- 目录
- 15.6 小结
- 16.1 上下文

上下文

每个流的上下文包含两种内部消息类型. 首先最常用的是上下文选项. 这些值被安排在上下文中一个二维数组中, 通常用于改变流包装器的初始化行为. 还有一种则是上下文参数, 它对于包装器是未知的, 当前提供了一种方式用于在流包装层内部的事件通知.

```
php_stream_context *php_stream_context_alloc(void);
```

通过这个API调用可以创建一个上下文, 它将分配一些存储空间并初始化用于保存上下文选项和参数的HashTable. 还会自动的注册为一个请求终止后将被清理的资源.

设置选项

设置上下文选项的内部API和用户空间的API是等同的:

```
int php_stream_context_set_option(php_stream_context *context,
            const char *wrappername, const char *optionname,
            zval *optionvalue);
```

下面是用户空间的原型:

```
bool stream_context_set_option(resource $context,
            string $wrapper, string $optionname,
            mixed $value);
```

它们的不同仅仅是用户空间和内部需要的数据类型不同.下面的例子就是使用这两个API调用, 通过内建包装器发起一个HTTP请求, 并通过一个上下文选项覆写了user_agent设置.

```
php_stream  *php_varstream_get_homepage(const char *alt_user_agent TSRMLS_DC)
{
    php_stream_context  *context;
    zval    tmpval;

    context = php_stream_context_alloc(TSRMLS_C);
    ZVAL_STRING(&tmpval, alt_user_agent, 0);
    php_stream_context_set_option(context, "http", "user_agent", &tmpval);
    return php_stream_open_wrapper_ex("http://www.php.net", "rb", REPORT_ERRORS | ENFORCE
}
```

> 译者使用的php-5.4.10中php_stream_context_alloc()增加了线程安全控制, 因此相应的对例子进行了修改, 请读者测试时注意. 这里要注意的是tmpval并没有分配任何持久性的存储空间, 它的字符串值是通过复制设置的. php_stream_context_set_option()会自动的对传入的zval内容进行一次拷贝.

取回选项

用于取回上下文选项的API调用正好是对应的设置API的镜像:

```
int php_stream_context_get_option(php_stream_context *context,
            const char *wrappername, const char *optionname,
            zval ***optionvalue);
```

回顾前面, 上下文选项存储在一个嵌套的HashTable中, 当从一个HashTable中取回值时, 一般的方法是传递一个指向zval **的指针给zend_hash_find(). 当然, 由于php_stream_context_get_option()是zend_hash_find()的一个特殊代理, 它们的语义是相同的.

下面是内建的http包装器使用php_stream_context_get_option()设置user_agent的简化版示例:

```
zval **ua_zval;
char *user_agent = "PHP/5.1.0";
if (context &&
    php_stream_context_get_option(context, "http",
                "user_agent", &ua_zval) == SUCCESS &&
                Z_TYPE_PP(ua_zval) == IS_STRING) {
    user_agent = Z_STRVAL_PP(ua_zval);
}
```

这种情况下, 非字符串值将会被丢弃, 因为对用户代理字符串而言, 数值是没有意义的. 其他的上下文选项, 比如max_redirects, 则需要数字值, 由于在字符串的zval中存储数字值并不通用, 所以需要执行一个类型转换以使设置合法.

不幸的是这些变量是上下文拥有的, 因此它们不能直接转换; 而需要首先进行隔离再进行转换, 最终如果需要还要进行销毁:

```
long max_redirects = 20;
zval **tmpzval;
if (context &&
    php_stream_context_get_option(context, "http",
            "max_redirects", &tmpzval) == SUCCESS) {
    if (Z_TYPE_PP(tmpzval) == IS_LONG) {
        max_redirects = Z_LVAL_PP(tmpzval);
    } else {
        zval copyval = **tmpzval;
        zval_copy_ctor(&copyval);
        convert_to_long(&copyval);
        max_redirects = Z_LVAL(copyval);
        zval_dtor(&copyval);
    }
}
```

> 实际上, 在这个例子中, zval_dtor()并不是必须的. IS_LONG的变量并不需要zval容器之外的存储空间, 因此zval_dtor()实际上不会有真正的操作. 在这个例子中包含它是为了完整性考虑, 对于字符串, 数组, 对象, 资源以及未来可能的其他类型, 就需要这个调用了.

参数

虽然用户空间API中看起来参数和上下文选项是类似的, 但实际上在语言内部的php_stream_context结构体中它们被定义为不同的成员.

目前只支持一个上下文参数: 通知器. php_stream_context结构体中的这个元素可以指向下面的php_stream_notifier结构体:

```
typedef struct {
    php_stream_notification_func func;
    void (*dtor)(php_stream_notifier *notifier);
    void *ptr;
    int mask;
    size_t progress, progress_max;
} php_stream_notifier;
```

当将一个php*stream_notifier*结构体赋值给*context->notifier*时, 它将提供一个回调函数*func*, 在特定的流上发生下表中的*PHP_STREAM_NOTIFY*代码表示的事件时被触发. 每个事件将会对应下面第二张表中的*PHPSTREAM_NOTIFY_SEVERITY*的级别:

事件代码	含义
RESOLVE	主机地址解析完成. 多数基于套接字的包装器将在连接之前执行这个查询.
CONNECT	套接字流连接到远程资源完成.
AUTH_REQUIRED	请求的资源不可用, 原因是访问控制以及缺失授权
MIME_TYPE_IS	远程资源的mime-type不可用
FILE_SIZE_IS	远程资源当前可用大小
REDIRECTED	原来的URL请求导致重定向到其他位置
PROGRESS	由于额外数据的传输导致php_stream_notifier结构体的progress以及(可能的)progress_max元素被更新(进度信息, 请参考php手册curl_setopt的CURLOPT_PROGRESSFUNCTION和CURLOPT_NOPROGRESS选项)
COMPLETED	流上没有更多的可用数据
FAILURE	请求的URL资源不成功或未完成
AUTH_RESULT	远程系统已经处理了授权认证

安全码	
INFO	信息更新. 等价于一个E_NOTICE错误
WARN	小的错误条件. 等价于一个E_WARNING错误
ERR	中断错误条件. 等价于一个E_ERROR错误.

通知器实现提供了一个便利指针*ptr用于存放额外数据. 这个指针指向的空间必须在上下文析构时被释放, 因此必须指定一个dtor函数, 在上下文的最后一个引用离开它的作用域时调用这个dtor进行释放.

mask元素允许事件触发限定特定的安全级别. 如果发生的事件没有包含在mask中, 则通知器函数不会被触发.

最后两个元素progress和progress_max可以由流实现设置, 然而, 通知器函数应该避免使用这两个值, 除非它接收到PHP_STREAM_NOTIFY_PROGRESS或PHP_STREAM_NOTIFY_FILE_SIZE_IS事件通知.

下面是一个php_stream_notification_func()回调原型的示例:

```
void php_sample6_notifier(php_stream_context *context,
        int notifycode, int severity, char *xmsg, int xcode,
        size_t bytes_sofar, size_t bytes_max,
        void *ptr TSRMLS_DC)
{
    if (notifycode != PHP_STREAM_NOTIFY_FAILURE) {
        /* 忽略所有通知 */
        return;
    }
    if (severity == PHP_STREAM_NOTIFY_SEVERITY_ERR) {
        /* 分发到错误处理函数 */
        php_sample6_theskyisfalling(context, xcode, xmsg);
        return;
    } else if (severity == PHP_STREAM_NOTIFY_SEVERITY_WARN) {
        /* 日志记录潜在问题 */
        php_sample6_logstrangeevent(context, xcode, xmsg);
        return;
    }
}
```

默认上下文

在php5.0中, 当用户空间的流创建函数被调用时, 如果没有传递上下文参数, 请求一般会使用默认的上下文. 这个上下文变量存储在文件全局结构中: FG(default_context), 并且它可以和其他所有的php_stream_context变量一样访问. 当在用户空间脚本执行流的创建时, 更好的方式是允许用户指定一个上下文或者至少指定一个默认的上下文. 将用户空间的zval *解码得到php_stream_context可以使用php_steram_context_from_zval()宏完成, 比如下面改编自第14章"访问流"的例子:

```
PHP_FUNCTION(sample6_fopen)
{
    php_stream *stream;
    char *path, *mode;
    int path_len, mode_len;
    int options = ENFORCE_SAFE_MODE | REPORT_ERRORS;
    zend_bool use_include_path = 0;
    zval *zcontext = NULL;
    php_stream_context *context;

    if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC,
            "ss|br", &path, &path_len, &mode, &mode_len,
                &use_include_path, &zcontext) == FAILURE) {
        return;
    }
    context = php_stream_context_from_zval(zcontext, 0);
    if (use_include_path) {
        options |= PHP_FILE_USE_INCLUDE_PATH;
    }
    stream = php_stream_open_wrapper_ex(path, mode, options,
```

```
                                          NULL, context);
    if (!stream) {
        RETURN_FALSE;
    }
    php_stream_to_zval(stream, return_value);
}
```

如果zcontext包含一个用户空间的上下文资源, 通过ZEND_FETCH_RESOURCE()调用获取到它关联的指针设置到context中. 否则, 如果zcontext为NULL并且php_stream_context_from_zval()的第二个参数设置为非0值, 这个宏则直接返回NULL. 这个例子以及几乎所有的核心流创建的用户空间函数中, 第二个参数都被设置为0, 此时将使用FG(default_context)的值.

links

- 目录
- 16 有趣的流
- 16.2 过滤器

过滤器

过滤器作为读写操作的流内容传输过程中的附加阶段. 要注意的是直到php 4.3中才加入了流过滤器, 在php 5.0对流过滤器的API设计做过较大的调整. 本章的内容遵循的是php 5的流过滤器规范.

在流上应用已有的过滤器

在一个打开的流上应用一个已有的过滤器只需要几行代码即可:

```
php_stream *php_sample6_fopen_read_ucase(const char *path
                                        TSRMLS_DC) {
    php_stream_filter *filter;
    php_stream *stream;

    stream = php_stream_open_wrapper_ex(path, "r",
                        REPORT_ERRORS | ENFORCE_SAFE_MODE,
                        NULL, FG(default_context));
    if (!stream) {
        return NULL;
    }

    filter = php_stream_filter_create("string.toupper", NULL,
                                        0 TSRMLS_CC);
    if (!filter) {
        php_stream_close(stream);
        return NULL;
    }
    php_stream_filter_append(&stream->readfilters, filter);

    return stream;
}
```

首先来看看这里引入的API函数以及它的兄弟函数:

```
php_stream_filter *php_stream_filter_create(
                const char *filtername, zval *filterparams,
                int persistent TSRMLS_DC);
void php_stream_filter_prepend(php_stream_filter_chain *chain,
                php_stream_filter *filter);
void php_stream_filter_append(php_stream_filter_chain *chain,
                php_stream_filter *filter);
```

php_stream_filter_create()的filterparams参数和用户空间对应的stream_filter_append()和stream_filter_prepend()函数的同名参数含义一致. 要注意, 所有传递到php_stream_filter_create()的zval *数据都不是过滤器所拥有的. 它们只是在过滤器创建期间被借用而已, 因此在调用作用域分配传入的所有内存空间都要手动释放.

如果过滤器要被应用到一个持久化流, 则必须设置persistent参数为非0值. 如果你不确认你要应用过滤器的流是否持久化的, 则可以使用php_stream_is_persistent()宏进行检查, 它只接受一个php_stream *类型的参

数.

如在前面例子中看到的, 流过滤器被隔离到两个独立的链条中. 一个用于写操作中对php_stream_write()调用响应时的stream->ops->write()调用之前. 另外一个用于读操作中对stream->ops->read()取回的所有数据进行处理.

在这个例子中你使用&stream->readfilters指示读的链条. 如果你想要在写的链条上应用一个过滤器, 则可以使用&stream->writefilters.

定义一个过滤器实现

注册过滤器实现和注册包装器遵循相同的基础规则. 第一步是在MINIT阶段向php中引入你的过滤器, 与之匹配的是在MSHUTDOWN阶段移除它. 下面是需要调用的API原型以及两个注册过滤器工厂的示例:

```
int php_stream_filter_register_factory(
            const char *filterpattern,
            php_stream_filter_factory *factory TSRMLS_DC);
int php_stream_filter_unregister_factory(
            const char *filterpattern TSRMLS_DC);

PHP_MINIT_FUNCTION(sample6)
{
    php_stream_filter_register_factory("sample6",
            &php_sample6_sample6_factory TSRMLS_CC);
    php_stream_filter_register_factory("sample.*",
            &php_sample6_samples_factory TSRMLS_CC);
    return SUCCESS;
}
PHP_MSHUTDOWN_FUNCTION(sample6)
{
    php_stream_filter_unregister_factory("sample6" TSRMLS_CC);
    php_stream_filter_unregister_factory("sample.*"
                                        TSRMLS_CC);
    return SUCCESS;
}
```

这里注册的第一个工厂定义了一个具体的过滤器名sample6; 第二个则利用了流包装层内部的基本匹配规则. 为了进行演示, 下面的用户空间代码, 每行都将尝试通过不同的名字实例化php_sample6_samples_factory.

```
<?php
    stream_filter_append(STDERR, 'sample.one');
    stream_filter_append(STDERR, 'sample.3');
    stream_filter_append(STDERR, 'sample.filter.thingymabob');
    stream_filter_append(STDERR, 'sample.whatever');
?>
```

php_sample6_samples_factory的定义如下面代码, 你可以将这些代码放到你的MINIT块上面:

```
#include "ext/standard/php_string.h"
```

```
typedef struct {
    char    is_persistent;
    char    *tr_from;
    char    *tr_to;
    int     tr_len;
} php_sample6_filter_data;

/* 过滤逻辑 */
static php_stream_filter_status_t php_sample6_filter(
        php_stream *stream, php_stream_filter *thisfilter,
        php_stream_bucket_brigade *buckets_in,
        php_stream_bucket_brigade *buckets_out,
        size_t *bytes_consumed, int flags TSRMLS_DC)
{
    php_stream_bucket       *bucket;
    php_sample6_filter_data *data       = thisfilter->abstract;
    size_t                  consumed    = 0;

    while ( buckets_in->head ) {
        bucket      = php_stream_bucket_make_writeable(buckets_in->head TSRMLS_CC);
        php_strtr(bucket->buf, bucket->buflen, data->tr_from, data->tr_to, data->tr_len);
        consumed    += bucket->buflen;
        php_stream_bucket_append(buckets_out, bucket TSRMLS_CC);
    }
    if ( bytes_consumed ) {
        *bytes_consumed = consumed;
    }
    return PSFS_PASS_ON;
}

/* 过滤器的释放 */
static void php_sample6_filter_dtor(php_stream_filter *thisfilter TSRMLS_DC)
{
    php_sample6_filter_data *data   = thisfilter->abstract;
    pefree(data, data->is_persistent);
}

/* 流过滤器操作表 */
static php_stream_filter_ops php_sample6_filter_ops = {
    php_sample6_filter,
    php_sample6_filter_dtor,
    "sample.*",
};

/* 字符翻译使用的表 */
#define PHP_SAMPLE6_ALPHA_UCASE     "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#define PHP_SAMPLE6_ALPHA_LCASE     "abcdefghijklmnopqrstuvwxyz"
#define PHP_SAMPLE6_ROT13_UCASE     "NOPQRSTUVWXYZABCDEFGHIJKLM"
#define PHP_SAMPLE6_ROT13_LCASE     "nopqrstuvwxyzabcdefghijklm"

/* 创建流过滤器实例的过程 */
static php_stream_filter *php_sample6_filter_create(
        const char *name, zval *param, int persistent TSRMLS_DC)
{
    php_sample6_filter_data *data;
    char                    *subname;

    /* 安全性检查 */
    if ( strlen(name) < sizeof("sample.") || strncmp(name, "sample.", sizeof("sample.") -
```

```
        return NULL;
    }

    /* 分配流过滤器数据 */
    data    = pemalloc(sizeof(php_sample6_filter_data), persistent);

    if ( !data ) {
        return NULL;
    }

    /* 设置持久性 */
    data->is_persistent = persistent;

    /* 根据调用时的名字, 对过滤器数据进行适当初始化 */
    subname = (char *)name + sizeof("sample.") - 1;
    if ( strcmp(subname, "ucase") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_LCASE;
        data->tr_to     = PHP_SAMPLE6_ALPHA_UCASE;
    } else if ( strcmp(subname, "lcase") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_UCASE;
        data->tr_to     = PHP_SAMPLE6_ALPHA_LCASE;
    } else if ( strcmp(subname, "rot13") == 0 ) {
        data->tr_from   = PHP_SAMPLE6_ALPHA_LCASE
                        PHP_SAMPLE6_ALPHA_UCASE;;
        data->tr_to     = PHP_SAMPLE6_ROT13_LCASE
                        PHP_SAMPLE6_ROT13_UCASE;
    } else {
        /* 不支持 */
        pefree(data, persistent);
        return NULL;
    }

    /* 节省未来使用时每次的计算 */
    data->tr_len    = strlen(data->tr_from);

    /* 分配一个php_stream_filter结构并按指定参数初始化 */
    return php_stream_filter_alloc(&php_sample6_filter_ops, data, persistent);
}

/* 流过滤器工厂, 用于创建流过滤器实例(php_stream_filter_append/prepend的时候) */
static php_stream_filter_factory php_sample6_samples_factory = {
    php_sample6_filter_create
};
```

> 译注: 下面是译者对整个流程的分析
>
> 一. MINIT阶段的register操作将在stream_filters_hash这个HashTable中注册一个php_stream_filter_factory结构, 它只有一个成员create_filter, 用来创建过滤器实例.
>
> 二. 用户空间代码stream_filter_append(STDERR, 'sapmple.one');在内部的实现是apply_filter_to_stream()函数(ext/standard/streamsfuncs.c中), 这里有两步操作, 首先创建过滤器, 然后将过滤器按照参数追加到流的readfilters/writefilters相应链中;
>
> 二.一 创建过滤器(php_stream_filter_create()): 首先直接按照传入的名字精确的从stream_filters_hash(或FG(stream_filters))中查找, 如果没有, 从右向左替换句点后面的内容为星

> 号"*"进行查找, 直到找到注册的过滤器工厂或错误返回. 一旦找到注册的过滤器工厂, 就调用它的create_filter成员, 创建流过滤器实例.
>
> 二.二 直接按照参数描述放入流的readfilters/writefilters相应位置.
>
> 三. 用户向该流进行写入或读取操作时(以写为例): 此时内部将调用_php_stream_write(), 在这个函数中, 如果流的writefilters非空, 则调用流过滤器的fops->filter()执行过滤, 并根据返回状态做相应处理.
>
> 四. 当流的生命周期结束, 流被释放的时候, 将会检查流的readfilters/writefilters是否为空, 如果非空, 相应的调用php_stream_filter_remove()进行释放, 其中就调用了fops->fdtor对流过滤器进行释放.

上一章我们已经熟悉了流包装器的实现, 你可能能够识别这里的基本结构. 工厂函数(php_sample6_samples_filter_create)被调用分配一个过滤器实例, 并赋值给一个操作集合和抽象数据. 这上面的例子中, 你的工厂为所有的过滤器类型赋值了相同的ops结构, 但使用了不同的初始化数据.

调用作用域将得到这里分配的过滤器, 并将它赋值给流的readfilters链或writefilters链. 接着, 当流的读/写操作被调用时, 过滤器链将数据放入到一个或多个php_stream_bucket结构体, 并将这些bucket组织到一个队列php_stream_bucket_brigade中传递给过滤器.

这里, 你的过滤器实现是前面的php_sample6_filter, 它取出输入队列bucket中的数据, 使用php_sample6_filter_create中确定的字符表执行字符串翻译, 并将修改后的bucket放入到输出队列.

由于这个过滤器的实现并没有其他内部缓冲, 因此几乎不可能出错, 因此它总是返回PSFS_PASS_ON, 告诉流包装层有数据被过滤器存放到了输出队列中. 如果过滤器执行了内部缓冲消耗了所有的输入数据而没有产生输出, 就需要返回PSFS_FEED_ME标识过滤器循环周期在没有其他输入数据时暂时停止. 如果过滤器碰到了关键性的错误, 它应该返回PSFS_ERR_FATAL, 它将指示流包装层, 过滤器链处于不稳定状态. 这将导致流被关闭.

用于维护bucket和bucket队列的API函数如下:

```
php_stream_bucket *php_stream_bucket_new(php_stream *stream,
                      char *buf, size_t buflen, int own_buf,
                      int buf_persistent TSRMLS_DC);
```

创建一个php_stream_bucket用于存放到输出队列. 如果own_buf被设置为非0值, 流包装层可以并且通常都会修改它的内容或在某些点释放分配的内存. buf_persistent的非0值标识buf使用的内存是否持久分配的:

```
int php_stream_bucket_split(php_stream_bucket *in,
        php_stream_bucket **left, php_stream_bucket **right,
        size_t length TSRMLS_DC);
```

这个函数将in这个bucket的内容分离到两个独立的bucket对象中. left这个bucket将包含in中的前length个字符, 而right则包含剩下的所有字符.

```
void php_stream_bucket_delref(php_stream_bucket *bucket
                                                 TSRMLS_DC);
void php_stream_bucket_addref(php_stream_bucket *bucket);
```

Bucket使用和zval以及资源相同的引用计数系统. 通常, 一个bucket仅属于一个上下文, 也就是它依附的队列.

```
void php_stream_bucket_prepend(
                    php_stream_bucket_brigade *brigade,
                    php_stream_bucket *bucket TSRMLS_DC);
void php_stream_bucket_append(
        php_stream_bucket_brigade *brigade,
        php_stream_bucket *bucket TSRMLS_DC);
```

这两个函数扮演了过滤器子系统的苦力, 用于附加bucket到队列的开始(prepend)或末尾(append)

```
void php_stream_bucket_unlink(php_stream_bucket *bucket
                                                 TSRMLS_DC);
```

在过滤器逻辑应用处理完成后, 旧的bucket必须使用这个函数从它的输入队列删除(unlink).

```
php_stream_bucket *php_stream_bucket_make_writeable(
        php_stream_bucket *bucket TSRMLS_DC);
```

将一个bucket从它所依附的队列中移除, 并且如果需要, 赋值bucket->buf的内部缓冲区, 这样就使得它的内容可修改. 在某些情况下, 比如当输入bucket的引用计数大于1时, 返回的bucket将会是不同的实例, 而不是传入的实例. 因此, 我们要保证在调用作用域使用的是返回的bucket, 而不是传入的bucket.

links

- 目录
- 16.1 上下文
- 16.3 小结

小结

过滤器和上下文可以让普通的流类型行为被修改, 或通过INI设置影响整个请求, 而不需要直接的代码修改. 使用本章设计的计数, 你可以使你自己的包装器实现更加强大, 并且可以对其他包装器产生的数据进行改变.

接下来, 我们将离开PHPAPI背后的工作, 回到php构建系统的机制, 产生更加复杂的扩展链接到其他应用, 找到更加容易的方法, 使用工具集处理重复的工作.

links

- 目录
- 16.2 过滤器
- 17 配置和链接

17 配置和链接

目录

所有前面示例中的代码, 都是你曾经在php用户空间编写过代码的C语言的独立版本. 如果你做的项目需要和php扩展进行粘合, 那么你就至少需要链接一个外部库.

links

- 目录
- 16.3 小结
- 17.1 autoconf

autoconf

在一个简单的应用中, 你可能已经在你的Makefile中增加了下面这样的CFLAGS和LDFLAGS.

```
CFLAGS = ${CFLAGS} -I/usr/local/foobar/include
LDFLAGS = ${LDFLAGS} -lfoobar -L/usr/local/foobar/lib
```

想要构建你的应用却没有libfoobar的人, 或将libfoobar安装到其他位置的人, 将会得到一个处理过的错误消息, 用于帮助他找到错误原因.

在过去十年开发的多数开发源代码软件(OSS)以及PHP都利用了一个实用工具autoconf, 通过一些简单的宏来生成复杂的configure脚本. 这个产生的脚本会执行查找依赖库已经头文件是否安装的工作. 基于这些信息, 一个包可以自定义构建代码行, 或在编译的时间被浪费之前提供一个有意义的错误消息.

在构建php扩展时, 无论你是否计划公开发布, 都需要利用这个autoconf机制. 即便你对autoconf已经很熟悉了, 也请花几分钟时间阅读本章, php中引入了一些一般安装的autoconf没有的自定义宏.

和传统的autoconf步骤(集中的configure.in文件包含了包的所有配置宏)不同, php只是用configure.in管理许多位域源码树下小的config.m4脚本的协调, 包括各个扩展, SAPI, 核心自身, 以及ZendEngine.

你已经在前面的章节看到了一个简单版本的config.m4. 接下来, 我们将在这个文件中增加其他的autoconf语法, 让你的扩展可以收集到更多的配置时信息.

links

- 目录
- 17 配置和链接
- 17.2 库的查找

库的查找

config.m4脚本最多是用于检查依赖库是否已安装. 扩展比如mysql, ldap, gmp以及其他设计为php用户空间和c库实现的其他功能之间的粘合层的扩展. 如果它们的依赖库没有安装, 或者安装的版本太旧, 要么会编译错误, 要么会导致产生的二进制无法运行.

头文件扫描

对依赖库扫描中最简单的一步就是检查你的脚本中的包含文件, 它们将在链接时使用. 下面的代码尝试在一些常见位置查找zlib.h:

```
PHP_ARG_WITH(zlib,[for zlib Support]
[  with-zlib              Include ZLIB Support])

if test "$PHP_ZLIB" != "no"; then
  for i in /usr /usr/local /opt; do
    if test -f $i/include/zlib/zlib.h; then
      ZLIB_DIR=$i
    fi
  done

  if test -z "$ZLIB_DIR"; then
    AC_MSG_ERROR([zlib not installed (http://www.zlib.org)])
  fi

  PHP_ADD_LIBRARY_WITH_PATH(z,$ZLIB_DIR/lib, ZLIB_SHARED_LIBADD)
  PHP_ADD_INCLUDE($ZLIB_DIR/include)

  AC_MSG_RESULT([found in $ZLIB_DIR])
  AC_DEFINE(HAVE_ZLIB,1,[libz found and included])

  PHP_NEW_EXTENSION(zlib, zlib.c, $ext_shared)
  PHP_SUBST(ZLIB_SHARED_LIBADD)
fi
```

config.m4文件很明显比你迄今为止使用的要大. 幸运的是, 它的语法非常的简单易懂并且如果你熟悉bash脚本, 对它也就不会陌生.

文件和第5章"你的第一个扩展"中第一次出现的一样, 都是以PHP_ARG_WITH()宏开始. 这个宏的行为和你用过的PHP_ARG_ENABLE()宏一样, 不过它将导致./configure时的选项是--with-extname/--without-extname而不再是--enable-extname/--disable-extname.

回顾这些宏, 它们的功能是等同的, 不同仅在于是否让终端用户给你的包一些暗示. 你可以在自己创建的私有扩展上使用任意一种方式. 不过, 如果你计划公开发布, 那就应该知道php正式的编码标准, 它指出enable/disable用于哪些不需要链接外部库的扩展, with/without则反之.

由于我们这里假设的扩展将链接zlib库, 因此你的config.m4脚本要以查找扩展源代码中将包含的zlib.h头文件. 这通过检查一些标准位置/usr, /usr/local, /opt中include/zlib目录下的zlib.h完成对其下两个目录的定位.

如果找到了zlib.h, 则将基路径设置到临时变量ZLIB_DIR中. 一旦循环完成, config.m4脚本检查ZLIB_DIR是否包含内容来确定是否找到了zlib.h. 如果没有找到, 则产生一个有意义的错误让用户知道./configure不能继续.

此刻, 脚本假设头文件存在, 对应的库也必须存在, 因此在下面的两行使用它修改构建环境, 最终增加-lz -L$ZLIB_DIR/lib到LDFLAGS以及-I$ZLIB_DIR/include到CFLAGS.

最终, 输出一个确认消息指示zlib安装已经找到, 并且在编译期间使用它的路径. config.m4的其他部分从前面部分的学习中你应该已经熟悉了. 为config.h定义一个#define, 定义扩展并指定它的源代码文件, 同时标识一个变量完成将扩展附加到构建系统的工作.

功能测试

迄今为止, 这个config.m4示例指示查找了需要的头文件. 尽管这已经够用了, 但它仍然不能确保产生的二进制正确的进行链接, 因为可能不存在匹配的库文件, 或者版本不正确.

最简单的检查zlib.h对应的libz.so库文件是否存在的方式就是检查文件是否存在:

```
if ! test -f $ZLIB_DIR/lib/libz.so; then
  AC_MSG_ERROR([zlib.h found, but libz.so not present!])
fi
```

当然, 这仅仅是问题的一面. 如果安装了其他的同名库但和你要查找的库不兼容怎么办呢? 确保你的扩展可以成功编译的最好方式是测试找到的库实际编译所需的内容. 要这样做就需要在config.m4中PHP_ADD_LIBRARY_WITH_PATH调用之前加入下面代码:

```
PHP_CHECK_LIBRARY(z, deflateInit,,[
  AC_MSG_ERROR([Invalid zlib extension, gzInit() not found])
],-L$ZLIB_DIR/lib)
```

这个工具宏将展开输出一个完整的程序, ./configure将尝试编译它. 如果编译成功, 表示第二个参数定义的符号在第一个参数指定的库中存在. 成功后, 第三个参数中指定的autoconf脚本将会执行; 失败后, 第四个参数中指定的autoconf脚本将执行. 在这个例子中, 第三个参数为空, 因为没有消息就是最好的消息(译注: 应该是unix哲学之一), 第五个参数也就是左后一个参数, 用于指定额外的编译器和链接器标记, 这里, 使用-L致命了一个额外的用于查找库的路径.

可选功能

那么现在你已经有正确的库和头文件了, 但依赖的是所安装库的哪个版本呢? 你可能需要某些功能或排斥某些功能. 由于这种类型的变更通常涉及到某些特定入口点的增加或删除, 因此可以重用PHP_CHECK_LIBRARY()宏来检查库的某些能力.

```
PHP_CHECK_LIBRARY(z, gzgets,[
  AC_DEFINE(HAVE_ZLIB_GETS,1,[Having gzgets indicates zlib >= 1.0.9])
],[
  AC_MSG_WARN([zlib < 1.0.9 installed, gzgets() will not be available])
```

```
],-L$ZLIB_DIR/lib)
```

测试实际行为

可能知道某个符号存在也还不能确保你的代码正确编译; 某些库的特定版本可能存在bug需要运行一些测试代码进行检查.

AC_TRY_RUN()宏可以编译一个小的源代码文件为可执行程序并执行. 依赖于传回给./configure的返回代码, 你的脚本可以设置可选的#define语句或直接输出消息(比如如果bug导致不能工作则提示升级)安全退出. 考虑下面的代码(摘自ext/standard/config.m4):

```
AC_TRY_RUN([
#include <math.h>

double somefn(double n) {
  return floor(n*pow(10,2) + 0.5);
}
int main() {
  return somefn(0.045)/10.0 != 0.5;
}
],[
  PHP_ROUND_FUZZ=0.5
  AC_MSG_RESULT(yes)
],[
  PHP_ROUND_FUZZ=0.50000000001
  AC_MSG_RESULT(no)
],[
  PHP_ROUND_FUZZ=0.50000000001
  AC_MSG_RESULT(cross compile)
])
AC_DEFINE_UNQUOTED(PHP_ROUND_FUZZ, $PHP_ROUND_FUZZ,
                    [Is double precision imprecise?])
```

你可以看到, AC_TRY_RUN()的第一个参数是一块C语言代码, 它将被编译执行. 如果这段代码的退出代码是0, 位于第二个参数的autoconf脚本将被执行, 这种情况标识round()函数和期望一样工作, 返回0.5.

如果代码块返回非0值, 位域第三个参数的autoconf脚本将被执行. 第四个参数(最后一个)在php交叉编译时使用. 这种情况下, 尝试运行示例代码是没有意义的, 因为目标平台不同于扩展编译时使用的平台.

links

- 目录
- 17.1 autoconf
- 17.3 强制模块依赖

强制模块依赖

在php 5.1中, 扩展之间的内部依赖是可以强制性的. 由于扩展可以静态构建到php中, 也可以构建为共享对象动态加载, 因此强制依赖需要在两个地方实现.

配置时模块依赖

第一个位置是你在本章课程中刚刚看到的config.m4文件中. 你可以使用PHP_ADD_EXTENSION_DEP(extname, depname[, optional])宏标识extname这个扩展依赖于depname这个扩展. 当extname以静态方式构建到php中时, ./configure脚本将使用这一行代码确认depname必须首先初始化. optional参数是一个标记, 用来标识depname如果也是静态构建的, 应该在extname之前加载, 不过它并不是必须的依赖.

这个宏的一个使用示例是pdo驱动, 比如pdo_mysql是可预知依赖于pdo扩展的:

```
ifdef([PHP_ADD_EXTENDION_DEP],
[
  PHP_ADD_EXTENSION_DEP(pdo_mysql, pdo)
])
```

要注意PHP_ADD_EXTENSION_DEP()宏被包裹到一个ifdef()结构中. 这是因为pdo和它的驱动在编译大于或等于5.0版本的php时都是存在的, 然而PHP_ADD_EXTENSION_DEP()宏是直到5.1.0版本才出现的.

运行时模块依赖

另外一个你需要注册依赖的地方是zend_module_entry结构体中. 考虑下面第5章中你定义的zend_module_entry结构体:

```
zend_module_entry sample_module_entry = {
#if ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE_EXTNAME,
    php_sample_functions,
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};
```

增加运行时模块依赖信息就需要对STANDARD_MOUDLE_HEADER部分进行一些小修改:

```
zend_module_entry sample_module_entry = {
#if ZEND_MODULE_API_NO >= 220050617
    STANDARD_MODULE_HEADER_EX, NULL,
    php_sample_deps,
#elif ZEND_MODULE_API_NO >= 20010901
    STANDARD_MODULE_HEADER,
#endif
    PHP_SAMPLE_EXTNAME,
    php_sample_functions,

    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
#if ZEND_MODULE_API_NO >= 20010901
    PHP_SAMPLE_EXTVER,
#endif
    STANDARD_MODULE_PROPERTIES
};
```

现在, 如果ZEND_MODULE_API_NO高于php 5.1.0 beta发布版, 则STANDARD_MODULE_HEADER(译注: 这里原著笔误为STANDARD_MODULE_PROPERTIES)将被替换为略微复杂的结构, 它将包含一个指向模块依赖信息的引用.

这个目标结构体可以在你的zend_module_entry结构体上面定义如下:

```
#if ZEND_MODULE_API_NO >= 220050617
static zend_module_dep php_sample_deps[] = {
    ZEND_MODULE_REQUIRED("zlib")
    {NULL,NULL,NULL}
};
#endif
```

和zend_function_entry向量类似, 这个列表可以有多项依赖, 按照顺序进行检查. 如果尝试加载某个依赖模块未满足, Zend将会中断加载, 报告不满足依赖的名字, 这样, 终端用户就可以通过首先加载其他模块来解决问题.

links

- 目录
- 17.2 库的查找
- 17.4 Windows方言

Windows方言

> 由于译者对windows环境不熟悉, 因此略过本节.

links

- 目录
- 17.3 强制模块依赖
- 17.5 小结

小结

如果你的扩展将在未知或不可控制的环境构建, 让它足够聪明以应付奇怪的环境就非常重要. 使用php提供的unix和windows上强有力的脚本能力, 你应该可以检测到麻烦并在未知的管理员需要电话求助之前给她一个解决方案.

现在你已经有使用php api从头建立php扩展的基础能力了, 你可以准备学习一下使用php提供的扩展开发工具把自己从繁重的重复劳动中解放出来了, 使用它们可以快速, 准确的建立新扩展的原型.

links

* 目录
* 17.4 Windows方言
* 18 扩展生成

18 扩展生成

目录

毫无疑问你已经注意到，每个php扩展都包含一些非常公共的并且非常单调的结构和文件。当开始一个新扩展开发的时候，如果这些公共的结构已经存在, 我们只用考虑填充功 能代码是很有意义的. 为此, 在php中包含了一个简单但是很有用的shell脚本。

links

- 目录
- 17.3 小结
- 18.1 ext_skel

ext_skel

切换到你的php源代码树下ext/目录中, 执行下面的命令:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ ./ext_skel extname=sample7
```

稍等便可, 输出一些文本, 你将看到下面的这些输出:

```
To use your new extension, you will have to execute the following steps: 1. $cd.. 2. $ vi
```

此刻观察ext/sample7目录, 你将看到在第5章"你的第一个扩展"中你编写的扩展骨架 代码的注释版本. 只是现在你还不能编译它; 不过只需要对config.m4做少许修改就可以让 它工作了, 这样你就可以避免第5章中你所做的大部分工作.生成函数原型

生成函数原型

如果你要编写一个对第三方库的包装扩展, 那么你就已经有了一个函数原型及基本行 为的机器刻度版本的描述(头文件), 通过传递一个额外的参数给./ext_skel, 它将自动的扫 描你的头文件并创建对应于接口的简单PHP_FUCNTION()块. 下面是使用./ext_skel指令 解析zlib头:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ ./ext_skel extname=sample8 \ proto=/usr/local/in
```

现在在ext/sample8/sample8.c中, 你就可以看到许多PHP_FUNCTION()定义, 每个 zlib函数对应一个. 要注意, 骨架生成程序会对某些未知资源类型产生警告消息. 你需要对 这些函数特别注意, 并且为了将这些内部的复杂结构体和用户空间可访问的变量关联起来, 可能会需要使用你在第9章"资源数据类型"中学到的知识.

links

- 目录
- 18 扩展生成
- 18.2 PECL_Gen

PECL_Gen

还有一种更加完善但也更加复杂的代码生成器: PECL_Gen, 可以在PECL(http:// pecl.php.net)中找到它, 使用pear install PECL_Gen命令可以安装它.

```
译者注: PECL_Gen已经迁移为CodeGen_PECL(http://pear.php.net/package/ CodeGen_PECL). 本章涉及代
```

一旦安装完成, 它就可以像ext_skel一样运行, 接受相同的输入参数, 产生大致相同的 输出, 或者如果提供了一个完整的xml定义文件, 则产生一个更加健壮和完整可编译版本的 扩展. PECL_Gen并不会节省你编写扩展核心功能的时间; 而是提供一种可选的方式高效 的生成扩展骨架代码.

specfile.xml

下面是最简单的扩展定义文件:

```
<?xml version="1.0" encoding="utf-8" ?> <extension name="sample9"> <functions>  <function
```

通过PECL_Gen命令运行这个文件:

```
jdoe@devbox:/home/jdoe/cvs/php-src/ext/$ pecl-gen specfile.xml
```

则会产生一个名为sample9的扩展, 并暴露一个用户空间函数sample9_hello_world().

关于扩展

除了你已经熟悉的功能文件, PECL_Gen还会产生一个package.xml文件 它可以用于 pear安装. 如果你计划发布包到PECL库, 或者哪怕你只是想要使用pear包系统交付内容, 有这个文件都会很有用.

总之, 你可以在PECL_Gen的specfile.xml中指定多数package.xml文件的元素.

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension name="sample9">
    <summary>Extension 9 generated by PECL_Gen</summary>
    <description>Another sample of PHP Extension Writing</description>
    <maintainers>
        <maintainer>
            <name>John D. Bookreader</name>
            <email>jdb@example.com</email>
            <role>lead</role>
        </maintainer>
    </maintainers>
    <release>
        <version>0.1</version>
        <date>2006-01-01</date>
        <state>beta</state>
```

```
        <notes>Initial Release</notes>
</release>
    ...
</extension>
```

当PECL_Gen创建扩展时, 这些信息将被翻译到最终的package.xml文件中. 依赖

依赖

如你在第17章"配置和链接"中所见, 依赖可以扫描出来用于config.m4和config.w32文 件. PECL_Gen可以使用定义各种类型的依赖完成扫描工作. 默认情况下, 列在 标签下的依赖会同时应用到Unix和win32构建中, 除非显式的是否用platform属性 指定某个目标

```
<?xml version="1.0" encoding="UTF-8" ?>
<extension name="sample9">
    ...
    <deps platform="unix">
        <! UNIX specific dependencies >
    </deps>
    <deps platform="win32">
        <! Win32 specific dependencies >
    </deps>
    <deps platform="all">
        <! Dependencies that apply to all platforms >
</deps>
    ...
</extension>
```

with

通常, 扩展在配置时使用--enable-extname样式的配置选项. 通过增加一个或多个 标签到块中, 则不仅配置选项被修改为--with-extname, 而且同时需要扫描 头文件:

```
<deps platform="unix">
    <with defaults="/usr:/usr/local:/opt"
        testfile="include/zlib/zlib.h">zlib headers</with>
</deps>
```

库

必须的库也列在下, 使用标签.

```
<deps platform="all">
    <lib name="ssleay" platform="win32"/>
    <lib name="crypto" platform="unix"/>
    <lib name="z" platform="unix" function="inflate"/>
</deps>
```

在前面两个例子中, 只是检查了库是否存在; 第三个例子中, 库将被真实的加载并扫描 以确认inflate()函数是

否定义.

尽管标签实际已经命名了目标平台, 但标签也有一个platform属性可以覆盖 标签的platform设置. 当它们混合使用的时候要格外小心.

此外, 需要包含的文件也可以通过在块中使用

标签在你的代码中追 加一个#include指令列表. 要强制某个头先包含, 可以在
标签上增加属性 prepend="yes". 和依赖类似,
也可以严格限制平台:

```
<deps>
    <header name="sys/types.h" platform="unix" prepend="yes"/>
    <header name="zlib/zlib.h"/>
</deps>
译注: 经测试, 译者的环境<header>标签不支持platform属性.
```

常量

用户空间常量使用块中的一个或多个标签定义. 每个标签需 要一个name和一个value属性, 以及一个值必须是int, float, string之一的type属性.

```
<constants>
    <constant name="SAMPLE9_APINO" type="int" value="20060101"/>
    <constant name="SAMPLE9_VERSION" type="float" value="1.0"/>
    <constant name="SAMPLE9_AUTHOR" type="string" value="John Doe"/>
</constants>
```

全局变量

线程安全全局变量的定义方式几乎相同. 唯一的不同在于type参数需要使用C语言原 型而不是php用户空间描述. 一旦定义并构建, 全局变量就可以使用第12章"启动, 终止, 以 及其中的一些点"中学习的EXTNAME_G(global_name)的宏用法进行访问. 在这里, value属性表示变量在请求启动时的默认值. 要注意在specfile.xml中这个默认值只能指定为简单 的标量数值. 字符串和其他复杂结构应该在RINIT阶段手动设置.

```
<globals>
    <global name="greeting" type="char *"/>
    <global name="greeting_was_issued" type="zend_bool" value="1"/>
</globals>
```

INI选项

要绑定线程安全的全局变量到php.ini设置, 则需要使用标签而不是. 这个标签需要两个额外的参数: onupdate="updatemethod"标识INI的修改应该怎样处理, access="mode"和第13章"INI设置"中介绍的模式含义相同, "mode"值可以是: all, user, perdir, system.

```
<globals>
    <phpini name="mysetting" type="int" value="42" onupdate="OnUpdateLong" access="all"/>
</globals>
```

函数

你已经看到了最基本的函数定义; 不过, 标签在PECL_Gen的specfile中实 际上支持两种不同类型的函数.

两个版本都支持你已经在级别上使用过的

和属 性; 两种类型都必须的元素是 `标签, 它包含了将要被放入你的源代码文件中的原文C语言代码.`

role="public"

如你所想, 所有定义为public角色的函数都将包装恰当的PHP_FUNCTION()头和花括 号, 对应到扩展的函数表向量中的条目.

除了其他函数支持的标签, public类型还允许指定一个标签. 这个标签的格式 应该匹配php在线手册中的原型展示, 它将被文档生成器解析.

```
<functions>
        <function role="public" name="sample9_greet_me">
            <summary>Greet a person by name</summary>
            <description>Accept a name parameter as a string and say hello to that person
Returns TRUE.</description>
            <proto>bool sample9_greet_me(string name)</proto>
            <code>
            <![CDATA[
            char *name;
            int name_len;
            if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "s",
                        &name, &name_len) == FAILURE) {
return; }
            php_printf("Hello ");
            PHPWRITE(name, name_len);
            php_printf("!\n");
            RETURN_TRUE;
]]>
            </code>
        </function>
    </functions>
```

role="internal"

内部函数涉及5个zend_module_entry函数: MINIT, MSHUTDOWN, RINIT, RSHUTDOWN, MINFO. 如果指定的名字不是这5个之一将会产生pecl-gen无法处理的错误.

```
<functions>
    <function role="internal" name="MINFO">
```

```
        <code>
        <![CDATA[
        php_info_print_table_start();
        php_info_print_table_header(2, "Column1", "Column2");
        php_info_print_table_end();
        ]]>
        </code>
    </function>
</functions>
```

自定义代码

所有其他需要存在于你的扩展中的代码都可以使用`标签包含. 要放置任意代码 到你的目标文件extname.c中, 使用role="code"; 或者说使用role="header"将代码放到目标 文件php_extname.h中. 默认情况下, 代码将放到代码或头文件的底部, 除非指定了 position="top"属性.`

```
<code role="header" position="bottom">
<![CDATA[
typedef struct _php_sample9_data {
    long val;
} php_sample9_data;
]]>
</code>
<code role="code" position="top">
<![CDATA[
static php_sample9_data *php_sample9_data_ctor(long value)
{
    php_sample9_data *ret;
    ret = emalloc(sizeof(php_sample9_data));
    ret->val = value;
    return ret;
}
]]> </code>
译注: 译者的环境中不支持原著中<code>标签的name属性.
```

links

- 目录
- 18.1 ext_skel
- 18.3 小结

小结

使用本章讨论的工具, 你就可以快速的开发php扩展, 并且让你的代码相比手写更加不 容易产生bug. 现在是时候转向将php嵌入到其他项目了. 剩下的章节中, 你将利用php环境 和强大的php引擎为你的已有项目增加脚本能力, 使它可以为你的客户提供更多更有用的 功能.

links

- 目录
- 18.2 PECL_Gen
- 19 设置宿主环境

19 设置宿主环境

目录

现在你已经了解了PHPAPI的世界, 并可以使用zval以及语言内部扩展机制执行很多 工作了, 是时候转移目标用它做它最擅长的事情了: 解释脚本代码.

links

- 目录
- 18.3 小结
- 19.1 嵌入式SAPI

嵌入式SAPI

回顾介绍中, php构建了一个层级系统. 最高层是提供用户空间函数和类库的所有扩 展. 同时, 其下是服务API(SAPI)层, 它扮演了webserver(比如apache, iis以及命令行接口 cli)的接口.

在这许多sapi实现中有一个特殊的sapi就是嵌入式sapi. 当这个sapi实现被构建时, 将 会创建一个包含所有你已知的php和zend api函数以及变量的库对象, 这个库对象还包含一些额外的帮助函数和宏, 用以简化外部程序的调用.

生成嵌入式api的库和头文件和其他sapi的编译所执行的动作相同. 只需要传递--enable-embed到./configure命令中即可. 和以前一样, 使用--enable-debug对于错误报告和 跟踪很有帮助.

你可能还需要打开--enable-maintainer-zts, 当然, 理由你已经耳熟能详了, 它将帮助 你注意到代码的错误, 不过, 这里还有其他原因. 假设某个时刻, 你有多个应用使用php嵌入 库执行脚本任务; 其中一个应用是简单的短生命周期的, 它并没有使用线程, 因此为了效率 你可能想要关闭ZTS.

现在假设第二个应用使用了线程, 比如webserver, 每个线程需要跟踪自己的请求上下 文. 如果ZTS被关闭, 则只有第一个应用可以使用这个库; 然而, 如果打开ZTS, 则两个应用 都可以在自己的进程空间使用同一个共享对象.

当然, 你也可以同时构建两个版本, 并给它们不同的名字, 但是这相比于在不需要ZTS 时包括ZTS带来的很小的效率影响更多的问题. 默认情况下, 嵌入式库将构建为libphp5.so共享对象, 或者在windows下的动态链接库, 不过, 它也可能使用可选的static关键字(--enable-embed=static)被构建为静态库.

构建为静态库的版本避免了ZTS/非ZTS的问题, 以及潜在的可能在一个系统中有多个 php版本的情况. 风险在于这就意味着你的结果应用二进制将显著变大, 它将承载整个 ZendEngine和PHP框架, 因此, 选择的时候就需要慎重的考虑你是否需要的是一个相对更小的库.

无论你选择那种构建方式,一旦你执行make install, libphp5都将被拷贝到你的./ configure指定的PREFIX目录下的lib/目录中. 此外还会在PREFIX/include/php/sapi/ embed目录下放入名为php_embed.h的头文件, 以及你在使用php嵌入式库编译程序时需 要的其他几个重要的头文件.

links

- 目录
- 19 设置宿主环境
- 19.2 构建并编译一个宿主应用

构建并编译一个宿主应用

究其本质而言, 库只是一个没有目的的代码集合. 为了让它工作, 你需要用以嵌入php 的应用. 首先, 我们来封装一个非常简单的应用, 它启动Zend引擎并初始化PHP处理一个请求, 接着就回头进行资源的清理.

```
#include <sapi/embed/php_embed.h>
int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc,argv)
    PHP_EMBED_END_BLOCK()
    return 0;
}
```

由于这涉及到了很多头文件, 构建实际上需要的时间要长于这么小的代码片段通常需 要的时间. 如果你使用了不同于默认路径(/usr/local)的PREFIX, 请确认以下面的方式指定 路径:

```
gcc -I /usr/local/php-dev/include/php/ \
    -I /usr/local/php-dev/include/php/main/ \
    -I /usr/local/php-dev/include/php/Zend/ \
    -I /usr/local/php-dev/include/php/TSRM/ \
    -lphp5 \
    -o embed1
    embed1.c
```

由于这个命令每次输入都很麻烦, 你可能更原意用一个简单的Makefile替代:

```
CC = gcc
CFLAGS = -c \
    -I /usr/local/php-dev/include/php/ \
    -I /usr/local/php-dev/include/php/main/ \
    -I /usr/local/php-dev/include/php/Zend/ \
    -I /usr/local/php-dev/include/php/TSRM/ \
    -Wall -g
LDFLAGS = -lphp5
all: embed1.c
    $(CC) -o embed1.o embed1.c $(CFLAGS)
    $(CC) -o embed1 embed1.o $(LDFLAGS)
```

这个Makefile和前面提供的命令有一些重要的区别. 首先, 它用-Wall开关打开了编译期的警 告, 并且用-g打开了调试信息. 此外它将编译和链接两个阶段分为了两个独立的阶段, 这样在后期 增加更多源文件的时候就相对容易. 请自己重新组着这个Makefile, 不过这里用于对齐的是Tab(水 平制表符)而不是空格.

现在, 你对embed1.c源文件做修改后, 只需要执行一一个make命令就可以构建出新的 embed1可执行程序了.

links

- 目录
- 19.1 嵌入式SAPI
- 19.3 通过嵌入包装重新创建cli

通过嵌入包装重新创建cli

现在php已经可以在你的应用中访问了, 是时候让它做一些事情了. 本章剩下的核心就是围绕着在这个测试应用框架中重新创建cli sapi展开的.

很简单, cli二进制程序最基础的功能就是在命令行指定一个脚本的名字, 由php对其解释执行. 用下面的代码替换你的embed1.c的内容就在你的应用中实现了cli.

```
#include <stdio.h>
#include <sapi/embed/php_embed.h>
int main(int argc, char *argv[]) {
    zend_file_handle    script;
/* 基本的参数检查 */ if ( argc <= 1 ) {
        fprintf(stderr, "Usage: %s <filename.php> <arguments>\n", argv[0]);
return -1; }
/* 设置一个文件处理结构 */
script.type
script.filename
script.opened_path
script.free_filename
if ( !(script.handle.fp = fopen(script.filename, "rb")) ) {
        fprintf(stderr, "Unable to open: %s\n", argv[1]);
return -1; }
/* 在将命令行参数注册给php时(php中的$argv/$argc), 忽略第一个命令行参数, 因为它对php脚本无意义 */
    argc --;
    argv ++;
    PHP_EMBED_START_BLOCK(argc, argv)
        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()
return 0; }
译注: 原著中的代码在译者的环境不能直接运行, 上面的代码是经过修改的.
```

当然, 你需要一个文件测试它, 创建一个小的php脚本, 命名为test.php, 在命令行使用你的embed程序执行它:

```
$ ./embed1 test.php
```

如果你给命令行传递了其他参数, 你可以在你的php脚本中使用$_SERVER['argc']/ $_SERVER['argv']看到它们.

你可能注意到了, 在PHP_EMBED_START_BLOCK()和PHP_EMBED_END_BLOCK()之间 的代码是缩进的. 这个细节是因为这两个宏实际上构成了一个C语言的代码块作用域. 也就是说PHP_EMBED_START_BLOCK()包含一个打开的花括号"{", 在PHP_EMBED_END_BLOCK()中 则有与之对应的关闭花括号"}". 这样做非常重要的一个问题是它们不能被放入到独立的启动/终止函数中. 下一章你将看到这个问题的解决方案.

links

- 目录

老技术新用

在PHP_EMBED_START_BLOCK()被调用后, 你的应用处于一个php请求周期的开始 位置, 相当于RINIT回调函数完成以后. 此刻你就可以和前面一样执行 php_execute_script()命令, 或者其他任意合法的, 可以在PHP_FUNCTION()或RINIT()块中出现的php/Zend API指令.

设置初始变量

第2章"变量的里里外外"中介绍了操纵符号表的概念, 第5至18章则介绍了怎样通过用 户空间脚本调用内部函数使用这些技术. 到这里这些处理也并没有发生变化, 虽然这里并 没有激活的用户空间脚本, 但是你的包装应用仍然可以操纵符号表. 将你的 PHP_EMBED_START_BLOCK()/PHP_EMBED_END_BLOCK()代码块替换为下面的代码:

```
PHP_EMBED_START_BLOCK(argc, argv)
    zval    *type;
    ALLOC_INIT_ZVAL(type);
    ZVAL_STRING(type, "Embedded", 1);
    ZEND_SET_SYMBOL(&EG(symbol_table), "type", type);
    php_execute_script(&script TSRMLS_CC);
PHP_EMBED_END_BLOCK()
```

现在使用make重新构建embed1, 并用下面的测试脚本进行测试:

```
<?php
    var_dump($type);
?>
```

当然, 这个简单的概念可以很容易的扩展为填充这个类型信息到$_SERVER超级全局变量数组中.

```
PHP_EMBED_START_BLOCK(argc, argv)
    zval    **SERVER_PP, *type;
    /* 注册$_SERVER超级全局变量 */
    zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
    /* 查找$_SERVER超级全局变量 */
    zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP)
    /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
    ALLOC_INIT_ZVAL(type);
    ZVAL_STRING(type, "Embedded", 1);
    ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);
    php_execute_script(&script TSRMLS_CC);
PHP_EMBED_END_BLOCK()
译注: 译者的环境中代码运行到zend_hash_find()处$_SERVER尚未注册, 经过跟踪, 发现它 是直到编译用户空间代
```

覆写INI选项

在第13章"INI设置"中, 有一部分是讲INI修改处理器的, 在那里看到的是INI阶段的处 理. PHP_EMBED_START_BLOCK()宏则将这些代码放到了运行时阶段. 也就是说这个时 候修改某些设置(比如

register_globals/magic_quotes_gpc)已经有点迟了.

不过在内部访问也没有什么不好. 所谓的"管理设置"比如safe_mode在这个略迟的阶 段可以使用下面的zend_alter_ini_entry()命令打开或关闭:

```
int zend_alter_ini_entry(char *name, uint name_length,
                          char *new_value, uint                new_value_length
                          int modify_type, int stage);
```

name, new_value以及它们对应的长度参数的含义正如你所预期的: 修改名为name的 INI设置的值为new_value. 要注意name_length包含了末尾的NULL字节, 然而 new_value_length则不包含; 然而, 无论如何, 两个字符串都必须是NULL终止的.

modify_type则提供简化的访问控制检查. 回顾每个INI设置都有一个modifiable属性, 它是PHP_INI_SYSTEM, PHP_INI_PERDIR, PHP_INI_USER等常量的组合值. 当使用 zend_alter_ini_entry()修改INI设置时, modify_type参数必须包含至少一个INI设置的 modifiable属性值.

用户空间的ini_set()函数通过传递PHP_INI_USER利用了这个特性, 也就是说只有 modifiable属性包含PHP_INI_USER标记的INI设置才能使用这个函数修改. 当在你的嵌入 式应用中使用这个API调用时, 你可以通过传递PHP_INI_ALL标记短路这个访问控制系统, 它将包含所有的INI访问级别.

stage必须对应于Zend Engine的当前状态; 对于这些简单的嵌入式示例, 总是 PHP_INI_STAGE_RUNTIME. 如果这是一个扩展或更高端的嵌入式应用, 你可能就需要将 这个值设置为PHP_INI_STAGE_STARTUP或PHP_INI_STAGE_ACTIVE.

下面是扩展embed1.c源文件, 让它在执行脚本文件之前强制开启safe_mode.

```
PHP_EMBED_START_BLOCK(argc, argv)
        zval    **SERVER_PP, *type;
/* 不论php.ini中如何设置都强制开启safe_mode */
        zend_alter_ini_entry("safe_mode", sizeof("safe_mode"), "1", sizeof("1") - 1, PHP_
PHP_INI_STAGE_RUNTIME);
/* 注册$_SERVER超级全局变量 */
zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
/* 查找$_SERVER超级全局变量 */
zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP) ;
        /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);
        php_execute_script(&script TSRMLS_CC);
PHP_EMBED_END_BLOCK()
```

定义附加的超级全局变量

在第12章"启动, 终止, 以及其中的一些点"中, 你知道了用户空间全局变量以及超级全 局变量可以在启动(MINIT)阶段定义. 同样, 本章介绍的嵌入式直接跳过了启动阶段, 处于 运行时状态. 和覆写INI一样, 这并不会显得太迟.

超级全局变量的定义实际上只需要在脚本编译之前定义即可, 并且在php的进程生命 周期中它只应该出现一次. 在扩展中的正常情况下, MINIT是唯一可以保证这些条件的地方.

由于你的包装应用现在是在控制中的, 因此可以保证定义用户空间自动全局变量的这 些点位于真正编译脚本源文件的php_execute_script()命令之前. 我们定义一个$_EMBED 超级全局变量并给它设置一个初始值来进行测试:

```
HP_EMBED_START_BLOCK(argc, argv)
        zval    **SERVER_PP, *type, *EMBED, *foo;
/* 在全局作用域创建$_EMBED数组 */ ALLOC_INIT_ZVAL(EMBED);
array_init(EMBED); ZEND_SET_SYMBOL(&EG(symbol_table), "_EMBED", EMBED);
        /* $_EMBED['foo'] = 'Bar'; */
        ALLOC_INIT_ZVAL(foo);
        ZVAL_STRING(foo, "Bar", 1);
        add_assoc_zval_ex(EMBED, "foo", sizeof("foo"), foo);
/* 注册超级全局变量$_EMBED */
        zend_register_auto_global("_EMBED", sizeof("_EMBED")
#ifdef ZEND_ENGINE_2
#else #endif
    , 1, NULL TSRMLS_CC);
    , 1 TSRMLS_CC);
/* 不论php.ini中如何设置都强制开启safe_mode */
        zend_alter_ini_entry("safe_mode", sizeof("safe_mode"), "1", sizeof("1") - 1, PHP_
PHP_INI_STAGE_RUNTIME);
/* 注册$_SERVER超级全局变量 */
zend_is_auto_global_quick("_SERVER", sizeof("_SERVER") - 1, 0 TSRMLS_CC);
/* 查找$_SERVER超级全局变量 */
zend_hash_find(&EG(symbol_table), "_SERVER", sizeof("_SERVER"), (void **)&SERVER_PP) ;
        /* $_SERVER['SAPI_TYPE'] = "Embedded"; */
        ALLOC_INIT_ZVAL(type);
        ZVAL_STRING(type, "Embedded", 1);
        ZEND_SET_SYMBOL(Z_ARRVAL_PP(SERVER_PP), "SAPI_TYPE", type);
        php_execute_script(&script TSRMLS_CC);
    PHP_EMBED_END_BLOCK()
```

要记住, Zend Engine 2(php 5.0或更高)使用了不同的zend_register_auto_global()元婴, 因此你需要用前面讲php 4兼容时候讲过的#ifdef. 如果你不关心旧版本php的兼容性, 则可以丢弃这些指令让代码变得更加整洁.

links

- 目录
- 19.3 通过嵌入包装重新创建cli
- 19.5 小结

小结

如你所见, 将完整的Zend Engine和PHP语言嵌入到你的应用中相比如扩展新功能来 说工作量要少. 由于它们共享相同的基础API, 我们可以学习尝试让其他实例可访问.

通过本章的学习, 你了解了最简单的嵌入式脚本代码格式, 同时还有all-in-one的宏PHP_EBED_START_BLOCK()和PHP_EMBED_END_BLOCK(). 下一章你将回到这些宏 的层的使用, 利用它们将php和你的宿主系统结合起来.

links

- 目录
- 19.4 老技术新用
- 20 高级嵌入式

20 高级嵌入式

目录

php的嵌入式能够提供的可不仅仅是同步的加载和执行脚本. 通过理解php的执行模块 各个部分是怎样组合的, 甚至给出一个脚本还可以回调到你的宿主应用中. 本章将涉及 SAPI层提供的I/O钩子带来的好处, 展开你已经从前面的主题中获取到信息的执行模块进行学习.

links

- 目录
- 19.5 小结
- 20.1 回调到php中

回调到php中

除了加载外部的脚本, 和你在上一章看到的类似, 你的php嵌入式应用, 下面将实现一个类似于用户空间eval()的命令.

```
int zend_eval_string(char *str, zval *retval_ptr,
                     char *string_name TSRMLS_DC)
```

这里, str是实际要执行的php脚本代码, 而string_name是一个与执行关联的任意描述信息. 如果发生错误, php会将这个描述信息作为错误输出中的"文件名". retval_ptr, 你应该 已经猜到了, 它将被设置为所传递代码产生的返回值. 试试用下面的代码创建新的项目吧.

```
#include <sapi/embed/php_embed.h>
int main(int argc, char *argv[]) {
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_eval_string("echo 'Hello World!';", NULL, "Simple Hello World App" TSRMLS_CC
    PHP_EMBED_END_BLOCK()
return 0;
}
```

现在使用命令或第19章"设置宿主环境"构建它(将Makefile中或命令中的embed1替换为embed2)

备选方案: 脚本文件的包含

可以预见的是, 这使得编译和执行外部脚本文件远比之前的方法更加容易, 因为你的 应用可以将原本复杂的打开/准备/执行的执行序列, 以这种简化但功能更加强大的设计替代:

```
#include <sapi/embed/php_embed.h>
int main(int argc, char *argv[]) {
    char    *filename;
    if ( argc <= 1 ) {
        fprintf(stderr, "Usage: %s <filename.php> <arguments>\n", argv[1]);
        return -1;
    }
    filename    = argv[1];
/* 忽略第0个参数 */ argc --;
argv ++;
    PHP_EMBED_START_BLOCK(argc, argv)
        char    *include_script;
        spprintf(&include_script, 0, "include '%s';", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    PHP_EMBED_END_BLOCK()
    return 0;
}
```

注意: 这种特殊的方法必须接受一个缺点, 如果文件名包含单引号, 将导致解析错误. 不过这可以通过使用

ext/standard/php_string.h中的php_addslashes()API调用解决. 花一些时间去阅读这个 文件以及附录中的API参考, 你会发现很多的特性, 它们可以让你避免在以后重造轮子.

调用用户空间函数

如你看到的加载和执行脚本文件, 在内部有两种方式调用用户空间函数. 现在最明显 的可能是重用zend_eval_string(), 将函数名和所有它的参数组织到一个庞大的字符串中, 然后收集返回值.

```
PHP_EMBED_START_BLOCK(argc, argv)
    char    *command;
    zval    retval;
    spprintf(&command, 0, "nl2br('%s');", argv[1]);
    zend_eval_string(command, &retval, "nl2br() execution" TSRMLS_CC);
    efree(command);
    printf("out: %s\n", Z_STRVAL(retval));
    zval_dtor(&retval);
PHP_EMBED_END_BLOCK()
```

和前面的include很像, 这个方法有一个致命的缺陷: 如果输入参数paramin(译者给出 的例子中是argv[1])给出一个错误的数据, 函数将会失败, 或者更糟糕的是导致无法预期的 结果. 解决方案是永远都避免编译代码的运行时片段, 并直接使用call_user_function()API调用函数.

```
int call_user_function(HashTable *function_table, zval **object_pp,
                       zval *function_name, zval *retval_ptr,
                       zend_uint param_count, zval *params[] TSRMLS_DC);
```

实际上从引擎外部调用时, function_table总是EG(function_table). 如果调用一个对象或类方法, object_pp需要是IS_OBJECT类型的调用实例zval, 或者对于类的静态调用则是 IS_STRING的值. function_name通常是IS_STRING的值, 包含要调用的函数名, 但是它也 可以是IS_ARRAY, 第0个元素包含一个对象或类名, 第1个元素包含方法名.

这个函数调用的结果是向传入的retval_ptr指向的zval设置返回值. param_count和 params扮演了argc/argv的角色. 也就是说, params[0]包含所传递的第一个参数, params[param_count - 1]包含了所传递的最后一个参数.

下面是用这种方法重新实现上面的例子:

```
PHP_EMBED_START_BLOCK(argc, argv)
    char    *command;
    zval    retval;
    spprintf(&command, 0, "nl2br('%s');", argv[1]);
    zend_eval_string(command, &retval, "nl2br() execution" TSRMLS_CC);
    efree(command);
    printf("out: %s\n", Z_STRVAL(retval));
    zval_dtor(&retval);
PHP_EMBED_END_BLOCK()
int call_user_function(HashTable *function_table, zval **object_pp,
                       zval *function_name, zval *retval_ptr,
                       zend_uint param_count, zval *params[] TSRMLS_DC);
PHP_EMBED_START_BLOCK(argc, argv)
```

```
    zval    *args[1];
    zval    retval, str, funcname;
    ZVAL_STRING(&funcname, "nl2br", 0);
    args[0] = &str;
    ZVAL_STRINGL(args[0], "HELLO WORLD!", sizeof("HELLO WORLD!"), 1);
    call_user_function(EG(function_table), NULL, &funcname, &retval, 1, args TSRMLS_CC);
    printf("out: %s\n", Z_STRVAL(retval));
    zval_dtor(args[0]);
    zval_dtor(&retval);
PHP_EMBED_END_BLOCK()
```

尽管代码看起来比较长, 但是工作量会显著降低, 因为这里没有要编译的中间代码, 传 递的数据不需要复制, 每个参数都已经在Zend兼容的结构体中. 同时, 要记得原来的例子中 在字符串中包含单引号时会有潜在的错误. 而这个版本没有这个问题.

links

- 目录
- 20 高级嵌入式
- 20.2 错误处理

错误处理

当发生错误时, 比如脚本解析错误, php将会进入到bailout模式. 在你已经看到的简单 的嵌入式例子中, 这表示它将直接跳到PHP_EMBED_END_BLOCK()宏, 并且绕过所有这个块中的剩余代码. 由于多数潜入php解释器的应用, 目的并不只是为了执行php代码, 因 此避免由于php脚本的故障导致整个应用崩溃是有意义的.

有一种方式可以将所有的执行限制到一个非常小的START/END块中, 这样发生崩溃 就只影响当前块. 这种方式的缺点是每个START/END块函数都是独立的PHP请求. 因此比 如下面START/END块, 虽然从语法逻辑上来看两个块是协同工作的, 但实际上它们之间是不共享公共作用域的.

```
int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_eval_string("$a = 1;", NULL, "Script Block 1");
    PHP_EMBED_END_BLOCK()
    PHP_EMBED_START_BLOCK(argc, argv)
/* 将打印出"NULL", 因为变量$a在这个请求中并没有定义. */
        zend_eval_string("var_dump($a);", NULL, "Script Block 2");
    PHP_EMBED_END_BLOCK()
    return 0;
}
```

还有一种解决方法是将两个zend_eval_string()调用使用Zend特有的伪语言结构 zend_try, zend_catch, zend_end_try进行隔离. 使用这些结构, 你的应用就可以按照想要的方式处理错误. 考虑下面的代码:

```
int main(int argc, char *argv[])
{
    PHP_EMBED_START_BLOCK(argc, argv)
        zend_try {
/* 尝试执行一一些可能失败的代码 */
            zend_eval_string("$1a = 1;", NULL, "Script Block 1a");
        } zend_catch {
/* 发生错误, 则尝试执行另外一一部分代码(一一般错误的补救或报告等行为) */
zend_eval_string("$a = 1;", NULL, "Script Block 1");
} zend_end_try();
/* 这里将显示"NULL", 因为变量$a在这个请求中没有定义. */ zend_eval_string("var_dump($a);", NULL,
    PHP_EMBED_END_BLOCK()
return 0; }
```

在这个示例的第二个版本中, zend_try块中将发生解析错误, 但它只影响自己的代码 块, 同时在zend_catch块中使用了一段好的代码对错误进行了处理. 同样你也可以尝试自 己给var_dump()部分也加上这些块.

```
译注: 这里对zend_try/zend_catch/zend_end_try解释的不是很清楚, 因此做以下补充说明. 读 者阅读这一部分
相关的定义如下:
#ifdef HAVE_SIGSETJMP#   define SETJMP(a) sigsetjmp(a, 0)
#   define LONGJMP(a,b) siglongjmp(a, b)
#   define JMP_BUF sigjmp_buf
#else
```

```
#   define SETJMP(a) setjmp(a)
#   define LONGJMP(a,b) longjmp(a, b)
#   define JMP_BUF jmp_buf
#endif
#define zend_try     \
{                    \
    JMP_BUF *__orig_bailout = EG(bailout);                   \
    JMP_BUF __bailout;                                       \
                                                                 \
    EG(bailout) = &__bailout;                                \
    if (SETJMP(__bailout)==0) {
#define zend_catch                                               \
        } else {                                                 \
            EG(bailout) = __orig_bailout;
#define zend_end_try() \ }\ EG(bailout) = __orig_bailout; \
}
```

zend_try{}代码块中的代码是在一个if语句中的, 这个if的条件是SETJMP(**bailout) == 0, SETJMP()**是在当前程序执行的点设置一个可回溯的点**(**保存了当前执行上下文和环境**), SETJMP()** 的返回比较特殊**,** 它有两种返回**: 1)** 直接返回**,** 此时返回值为**0; 2)** 调用**LONGJMP()**返回到对应 bailout当时调用SETJMP()的位置, 此时返回值非0.

基于上面的论述, 可以看出, 当zend_try的代码块中调用了LONGJMP()的时候, 程序将回到if (SETJMP(__bailout) == 0)的位置开始执行, 并且它的返回值为-1, 因此, 进入到对应的else语句 块, 也就是zend_catch语句块的代码.

zend_end_try()则只是一个结尾的花括号.

php中的这个伪语言结构正式这种方式实现的异常处理机制, 在系统的关键点调用 zend_bailout()(在Zend/zend.h中定义)即可.

本例中, 译者增加了zend_bailout()调用, 演示了这个伪语言结构的使用.

links

- 目录
- 20.1 回调到php中
- 20.3 初始化php

初始化php

迄今为止, 你看到的PHP_EMBED_START_BLOCK()和 PHP_EMBED_END_BLOCK()宏都用于启动, 执行, 终止一个紧凑的原子的php请求。 这样 做的优点是任何导致php bailout的错误顶多影响到PHP_EMBED_END_BLOCK()宏之内 的当前作用域. 通过将你的代码执行放入到这两个宏之间的小块中, php的错误就不会影响到你的整个应用. 你刚才已经看到了, 这种短小精悍的方法主要的缺点在于每次你建立一个新的 START/END块的时候, 都需要创建一个新的请求, 新的符号表, 因此就失去了所有的持久性语义. 要想同时得到两种优点(持久化和错误处理), 就需要将START和END宏分解为它们各 自的组件(译注: 如果不明白可以参考这两个宏的定义). 下面是本章开始给出的embed2.c 程序, 这一次, 我们对它进行了分解:

```
#include <sapi/embed/php_embed.h>
int main(int argc, char *argv[])
{
    #ifdef ZTS
    void ***tsrm_ls;
    #endif

    php_embed_init(argc, argv PTSRMLS_CC);
    zend_first_try {
        zend_eval_string("echo 'Hello World!';", NULL,"Embed 2 Eval'd string" TSRMLS_CC);
    } zend_end_try();
    php_embed_shutdown(TSRMLS_C);
    return 0;
}
```

它执行和之前一样的代码, 只是这一次你可以看到打开和关闭的括号包裹了你的代码, 而不是无法分开的START和END块。 将php_embed_init()放到你应用的开始, 将php_embed_shutdown()放到末尾, 你的应用就得到了一个持久的单请求生命周期, 它还可 以使用zend_first_try {} zend_end_try(); 结构捕获所有可能导致你整个包装应用跳出末尾 的PHP_EMBED_END_BLOCK()宏的致命错误. 为了看看真实世界环境的这种方法的应用, 我们将本章前面一些的例子的启动和终止 处理进行了抽象:

```
#include <sapi/embed/php_embed.h>
#ifdef ZTS
void ***tsrm_ls;
#endif
static void startup_php(void)
{
    /* Create "dummy" argc/argv to hide the arguments
     * meant for our actual application */
     int argc = 1;
     char *argv[2] = { "embed4", NULL };
     php_embed_init(argc, argv PTSRMLS_CC);
}

static void shutdown_php(void)
{
    php_embed_shutdown(TSRMLS_C);
}

static void execute_php(char *filename) {
```

```
    zend_first_try {
        char *include_script;
        spprintf(&include_script, 0, "include '%s';", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    } zend_end_try();
}

int main(int argc, char *argv[])
{
    if (argc <= 1) {
        printf("Usage: embed4 scriptfile");
        return -1;
    }
    startup_php();
    execute_php(argv[1]);
    shutdown_php();
    return 0;
}
```

类似的概念也可以应用到处理任意代码的执行以及其他任务. 只需要确认在最外部的 容器上使用zend_first_try, 则里面的每个容器上使用zend_try即可.

links

- 目录
- 20.2 错误处理
- 20.4 覆写INI_SYSTEM和INI_PERDIR选项

覆写INI_SYSTEM和INI_PERDIR选项

在上一章中, 你曾经使用zend_alter_ini_setting()修改过一些php的ini选项. 由于samp/embed直接将你的脚本推入了运行时模式, 因此许多重要的INI选项在控制返回到你的应用 时并没有被修改. 为了修改这些值, 就需要在主引擎启动之后而请求启动之前执行代码.

有一种方式是拷贝php_embed_init()的内容到你的应用中, 在你的本地拷贝中做必要 的修改, 接着使用你修改后的版本替代它. 当然这种方式可能会有问题.

首先也是最重要的, 你实际已经对别人的部分代码做了分支, 然而可能别人还会向其 中添加新的代码. 现在, 你就不再是只维护自己的应用了, 还需要保持分支出来的代码和主 分支保持一致. 幸运的是, 还有几种更简单的方法:

覆写默认的php.ini文件

因为嵌入式和其他的php sapi实现一样都是sapi, 它通过一个sapi_module_struct挂入 到引擎中. 嵌入式SAPI定义并设置了这个结构体的一个实例, 你的应用可以在调用 php_embed_init()之前访问它.

在这个结构体中, 有一个名为php_ini_path_override的char *类型字段. 为了让嵌入的 请求使用你的可选文件扩展php和Zend, 只需要在调用php_embed_init()之前将这个字段 设置为NULL终止的字符串. 下面是embed4.c中修改版的startup_php()函数:

```
static void startup_php(void)
{
    /* Create "dummy" argc/argv to hide the arguments
     * meant for our actual application */
    int argc = 1;
    char *argv[2] = { "embed4", NULL };
    php_embed_module.php_ini_path_override = "/etc/php_embed4.ini";
    php_embed_init(argc, argv PTSRMLS_CC);
}
```

这就使得每个使用嵌入库的应用可以保持自定义, 而不用将自己的配置暴露给别人. 相反, 如果你想要你的应用不使用php.ini, 只需要设置php_embed_module的 php_ini_ignore字段, 这样所有的设置都将使用内建的默认值, 除非由你的应用手动进行修改.

覆写嵌入启动

sapi_module_struct结构还包含一些回调函数, 下面是其中4个在PHP启动和终止阶段 比较有用的回调:

```
/* From main/SAPI.h */
typedef struct _sapi_module_struct {
    ...
    int (*startup)(struct _sapi_module_struct *sapi_module);
    int (*shutdown)(struct _sapi_module_struct *sapi_module);
    int (*activate)(TSRMLS_D);
    int (*deactivate)(TSRMLS_D);
    ...
} sapi_module_struct;
```

这些方法的名字熟悉吗? 它们对应于扩展的MINIT, MSHUTDOWN, RINIT, RSHUTDOWN, 并且和对应在扩展生命周期中的阶段一致. 要利用这些钩子, 可以如下修 改embed4中的startup_php()函数:

```
static int (*original_embed_startup)(struct _sapi_module_struct *sapi_module);
static int embed4_startup_callback(struct _sapi_module_struct *sapi_module)
{
/* 首先调用原来的启动回调, 否则环境未就绪 */
if (original_embed_startup(sapi_module) == FAILURE) {
/* 这里可以做应用的失败处理 */
        return FAILURE;
    }
/* 调用原来的embed_startup实际上让我们进入到ACTIVATE阶段而不是STARTUP阶段, * 但是我们仍然可以修改多
*/
    zend_alter_ini_entry("max_execution_time", sizeof("max_execution_time"),
                    "15", sizeof("15") - 1, PHP_INI_SYSTEM, PHP_INI_STAGE_ACTIVATE);
    zend_alter_ini_entry("safe_mode", sizeof("safe_mode"),
                    "1", sizeof("1") - 1, PHP_INI_SYSTEM, PHP_INI_STAGE_ACTIVATE);
    return SUCCESS;
}
static void startup_php(void)
{
    /* 创建假的argc/argv, 隐藏应用实际的参数 */ int argc = 1;
    char *argv[2] = { "embed4", NULL };
    /* 使用我们自己的启动函数覆写标准的启动方法, 但是保留了原来的指针, 因此它仍然能够被调用到 */ origir
    php_embed_module.startup = embed4_startup_callback;
    php_embed_init(argc, argv PTSRMLS_CC);
}
```

使用safe_mode, open_basedir这样的选项, 以及其他用以限制独立脚本行为的选项, 可以让你的应用更加安全可靠.

links

- 目录
- 20.3 初始化php
- 20.5 捕获输出

捕获输出

除非你开发的是非常简单的控制台应用, 否则你应该不希望php脚本代码产生的输出 直接被扔到激活的终端上. 捕获这些输出和你刚才用以覆写启动处理器的方法类似.

在sapi_module_struct中还有一些有用的回调:

```
typedef struct _sapi_module_struct {
    ...
    int (*ub_write)(const char *str, unsigned int str_length TSRMLS_DC);
    void (*flush)(void *server_context);
    void (*sapi_error)(int type, const char *error_msg, ...);
    void (*log_message)(char *message);
    ...
} sapi_module_struct;
```

标准输出: ub_write

所有用户空间的echo和print语句产生的输出, 以及其他内部通过php_printf()或 PHPWRITE()产生的输出, 最终都将被发送到激活的SAPI的ub_write()方法. 默认情况, 嵌入式SAPI直接将这些数据交给stdout管道, 而不关心你的应用的输出策略.

假设你的应用想要把所有的输出都发送到一个独立的控制台窗口; 你可能需要实现一个类似于下面伪代码块所描述的回调:

```
static int embed4_ub_write(const char *str, unsigned int str_length TSRMLS_DC)
{
    output_string_to_window(CONSOLE_WINDOW_ID, str, str_length);
    return str_length;
}
```

要让这个函数能够处理php产生的内容, 你需要在调用php_embed_init()之前对 php_embed_module结构做适当的修改:

```
php_embed_module.ub_write = embed4_ub_write;
```

注意: 哪怕你决定你的应用不需要php产生的输出, 也必须为ub_write设置一个回调. 将它的值设置为NULL将导致引擎崩溃, 当然, 你的应用也不能幸免.

缓冲输出: Flush

你的应用可能会使用缓冲php产生的输出进行优化, sapi层提供了一个回调用以通知 你的应用"现在请发送你的缓冲区数据", 你的应用并没有义务去实施这个通知; 不过, 由于 这个信息通常是由于足够的理由(比如到达请求结束位置)才产生的, 听从这个意见并不会有什么坏处.

下面的这对回调函数, 以256字节缓冲区缓冲数据由引擎安排执行flush.

```
char buffer[256];
int buffer_pos = 0;
static int embed4_ubwrite(const char *str, unsigned int str_length TSRMLS_DC)
{
    char *s = str;
char *d = buffer + buffer_pos;
int consumed = 0;
/* 缓冲区够用, 直接追加到缓冲区后面 */
if (str_length < (256 - buffer_pos)) {
        memcpy(d, s, str_length);
        buffer_pos += str_length;
        return str_length;
}
consumed = 256 - buffer_pos; memcpy(d, s, consumed); embed4_output_chunk(buffer, 256); st
s += consumed;
/* 消耗整个传入的块 */
while (str_length >= 256) {
        embed4_output_chunk(s, 256);
        s += 256;
        consumed += 256;
}
/* 重置缓冲区头指针内容 */ memcpy(buffer, s, str_length); buffer_pos = str_length; consumed 
return consumed;
}
static void embed4_flush(void *server_context)
{
if (buffer_pos < 0) {
/* 输出缓冲区中剩下的内容 */ embed4_output_chunk(buffer, buffer_pos); buffer_pos = 0;
}
}
```

在startup_php()中增加下面的代码, 这个基础的缓冲机制就就绪了:

```
php_embed_module.ub_write = embed4_ub_write;
php_embed_module.flush    = embed4_flush;
```

标准错误: log_message

在启用了log_errors INI设置时, 在启动或执行脚本时如果碰到错误, 将激活 log_message回调. 默认的php错误处理程序会在处理显示(这里是调用log_message回调)之前, 格式化这些错误消息, 使其称为整齐的, 人类可读的内容.

关于log_message回调, 这里你需要注意的第一件事是它并不包含长度参数, 因此它并不是二进制安全的. 也就是说, 它只是按照NULL终止来处理字符串末尾.

使用它来做错误报告通常不会有什么问题, 实际上, 它可以用于在错误消息的呈现上 做更多的事情. 默认情况下, sapi/embed将会通过这个简单的内建回调, 发送这些错误消息到标准错误管道:

```
static void php_embed_log_message(char *message)
{
    fprintf (stderr, "%s\n", message);
```

```
}
```

如果你想发送这些消息到日志文件, 则可以使用下面的版本替代:

```
static void embed4_log_message(char *message)
{
    FILE *log;
    log = fopen("/var/log/embed4.log", "a");
    fprintf (log, "%s\n", message);
    fclose(log);
}
```

特殊错误: sapi_error

少数特殊情况的错误属于某个sapi, 因此将绕过php的主错误处理程序. 这些错误一般 是由于使用不当造成的, 比如非web应用不应该使用header()函数, 上传文件到控制台应用程序等.

由于这些情况都离你所开发的sapi/embed应用非常遥远, 因此最好保持这个回调为空. 不过, 如果你非要坚持去捕获每种类型错误的源, 也只需要实现一个回调函数, 并在调 用php_embed_init()之前覆写它就可以了.

links

- 目录
- 20.4 覆写INI_SYSTEM和INI_PERDIR选项
- 20.6 同时扩展和嵌入

同时扩展和嵌入

在你的应用中运行php代码固然不错, 但是此刻, php执行环境仍然和你的主应用是隔离的, 它们并没有在真正意义上的一个层级进行交互.

现在你应该对php扩展的开发以及构建启用方面比较熟悉了. 你也已经有完成了嵌入 工作的例程, 这样就省去了这份工作. 将扩展代码植入到嵌入式应用中的工作量要比标准扩展小. 下面是一个新的嵌入式项目:

```
#include <sapi/embed/php_embed.h>
#ifdef ZTS
    void ***tsrm_ls;
#endif
/* Extension bits */
zend_module_entry php_mymod_module_entry = {
    STANDARD_MODULE_HEADER,
    "mymod", /* extension name */
    NULL, /* function entries */
    NULL, /* MINIT */
    NULL, /* MSHUTDOWN */
    NULL, /* RINIT */
    NULL, /* RSHUTDOWN */
    NULL, /* MINFO */
    "1.0", /* version */
    STANDARD_MODULE_PROPERTIES
};
/* Embedded bits */
static void startup_php(void)
{
    int argc = 1;
    char *argv[2] = { "embed5", NULL };
    php_embed_init(argc, argv PTSRMLS_CC);
    zend_startup_module(&php_mymod_module_entry);
}
static void execute_php(char *filename)
{
    zend_first_try {
        char *include_script;
        spprintf(&include_script, 0, "include '%s'", filename);
        zend_eval_string(include_script, NULL, filename TSRMLS_CC);
        efree(include_script);
    } zend_end_try();
]
int main(int argc, char *argv[])
{
    if (argc <= 1) {
        printf("Usage: embed4 scriptfile";);
        return -1;
    }
    startup_php();
    execute_php(argv[1]);
    php_embed_shutdown(TSRMLS_CC);
    return 0;
}
```

现在, 你可以定义function_entry向量, 启动/终止函数, 定义类, 以及所有你想增加的东 西. 现在, 它和你使用用户空间的dl()命令加载这个扩展库一样, 在这一个命令中Zend将自 动的处理所有的钩子并对你的模块进行注册, 就绪等待使用.(译注: startup_php()中调用 zend_startup_module(&php_mymod_module_entry)进行了模块注册)

links

- 目录
- 20.5 捕获输出
- 20.7 小结

小结

本章你看了一些上一章的一些简单的嵌入式示例进行了扩展, 你已经可以将php放入 到各种非线程应用了. 现在你已经掌握了扩展和嵌入式的基础, 并且可以在zval, 类, 资源, HashTable上工作了, 你已经可以真正开始一个真正的项目了.

在剩下的附录中, 你将看到php, zend以及其他扩展暴露的很多API函数. 你将会看到一些常用的代码片段以及近几年数以百计的开源PECL项目, 它们都可以作为你未来项目 的参考.

links

- 目录
- 20.6 同时扩展和嵌入